NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

<div>

SOFTWARE COMPONENTS FOR AIR DEFENSE PLANNING

by

Arent Arntzen

September 1998

Thesis Advisor: Arnold H. Buss
Second Reader: Gordon H. Bradley

</div>

Approved for public release; distribution is unlimited.

19981110 136

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1998	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE SOFTWARE COMPONENTS FOR AIR DEFENSE PLANNING			**5. FUNDING NUMBERS** N0001498WR20001
6. AUTHOR(S) Arent Arntzen			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Office of Scientific Research, 110 Duncan Avenue, Suite 100, Bolling AFB, DC 20332-0001			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release, distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (Maximum 200 words)

Modern offensive weapon technologies such as stealth and precision guided munitions have rendered Integrated Air Defense Systems increasingly vulnerable and ineffective. Stealth effectively reduces the performance of radar, but does not have the same impact on passive systems. Sensors have been the most important and vulnerable part of air defense systems throughout the history of air warfare. Research into passive sensors has been encouraging, but before passive sensor systems are produced, procured and deployed, analysis and planning must be conducted to quantify potential benefit and determine feasible system configurations. As this type of analysis encompasses extremely complex system behavior, developing reusable and flexible simulation models becomes important. This thesis develops a prototype software component architecture and component library for building simulation models for air defense analysis. Sensor and airborne weapon simulation components are demonstrated and used in an exploratory analysis of the impact of a network of Infrared Search and Track sensors. The analysis is based on a modern air defense system deployed in a realistic scenario. The component architecture and documentation methodology supports reuse, and provides model configuration flexibility with potential for growth in successive stages of analysis.

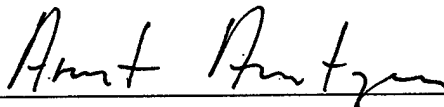| **14. SUBJECT TERMS** Air Defense Planning, Simulation, Software Components			**15. NUMBER OF PAGES** 100
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

SOFTWARE COMPONENTS FOR AIR DEFENSE PLANNING

Arent Arntzen
Major, Royal Norwegian Air Force
Submitted in partial fulfillment of the
requirements for the degree of

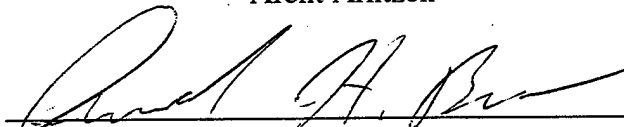## MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

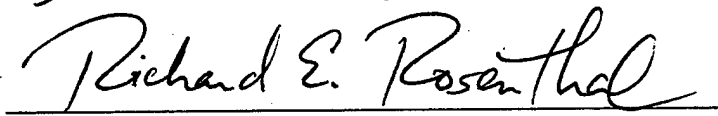## NAVAL POSTGRADUATE SCHOOL
September 1998

Author: _____
                    Arent Arntzen

Approved by: _____
                    Arnold H. Buss, Thesis Advisor

_____
                    Gordon H. Bradley, Second Reader

_____
                    Richard E. Rosenthal, Chairman
                    Department of Operations Research

ABSTRACT

Modern offensive weapon technologies such as stealth and precision guided munitions have rendered Integrated Air Defense Systems increasingly vulnerable and ineffective. Stealth effectively reduces the performance of radar, but does not have the same impact on passive systems. Sensors have been the most important and vulnerable part of air defense systems throughout the history of air warfare. Research into passive sensors has been encouraging, but before passive sensor systems are produced, procured and deployed, analysis and planning must be conducted to quantify potential benefit and determine feasible system configurations. As this type of analysis encompasses extremely complex system behavior, developing reusable and flexible simulation models becomes important. This thesis develops a prototype software component architecture and component library for building simulation models for air defense analysis. Sensor and airborne weapon simulation components are demonstrated and used in an exploratory analysis of the impact of a network of Infrared Search and Track sensors. The analysis is based on a modern air defense system deployed in a realistic scenario. The component architecture and documentation methodology supports reuse, and provides model configuration flexibility with potential for growth in successive stages of analysis.

THESIS DISCLAIMER

The reader is cautioned that the computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

TABLE OF CONTENTS

EXECUTIVE SUMMARY

Modern offensive weapon technologies such as stealth and precision guided munitions have rendered Integrated Air Defense Systems increasingly vulnerable and ineffective. Since air defense is a purely reactive form of warfare, the application of scientific principles to the design and deployment of air defense systems is a major factor in achieving effectiveness. Today's air defense planners face rapidly changing technological developments, both for offensive weapons and for sensors. Understanding the impact of technology on air defense operations must be done continually and at an increasing pace. The combination of dwindling defense resources and rapid technological developments makes the need for analysis more critical. Yet with current software architectures, even the analysis activity may be prohibitively costly for small nations.

Stealth effectively reduces the performance of radar, but does not have the same impact on passive systems. Sensors have been the most important and vulnerable part of air defense systems throughout the history of air warfare. Research into passive sensors has been encouraging, but before passive sensor systems are produced, procured and deployed, analysis and planning must be conducted to quantify potential benefit and determine feasible system configurations. As this type of analysis encompasses extremely complex system behavior, developing reusable and flexible models becomes important.

Of all modeling tools available, system simulation is perhaps the only one capable of capturing the behavior of Integrated Air Defense Systems. Unfortunately, building and using a simulation model is an expensive, slow, and cumbersome activity. Since model abstractions must ultimately be turned into computer code, the productivity of simulation modeling depends heavily on effective software engineering and programming.

Reuse is the key to increasing effectiveness in simulation modeling. Component Software is a technology that allows reuse of both model abstractions and implementations. Furthermore, this technology makes the simulation model scalable,

allowing the analyst to start with a simple model and build towards higher complexity and fidelity. Building models using software components thus allows the analyst to develop a model in a series of stepwise refinements. Progressing in small steps using components, the analyst can derive the simplest possible model for the task at hand, minimizing the effort that goes into parts of the model that ultimately would not be used thus increasing productivity.

This thesis uses Java™, a new and powerful object-oriented programming language, to develop a prototype software component architecture and component library for building simulation models for air defense. Sensor and airborne weapon simulation components are demonstrated and used in an exploratory analysis of the impact of a network of Infrared Search and Track sensors. A practical scenario comprising a modern medium range Surface to Air System (MSAM) is laid out as the basis for the simulation models. The data gathered from the models indicate that IRST systems could be valuable in the near future.

High tempo seems to be a dominating feature of theories of modern warfare. If simulation models are to be used for planning purposes under such circumstances the cycle time from one model to the next must be very short. Current methods fall far short of this requirement. In addition to providing model configuration flexibility and scalability, the component architecture supports reuse and makes data collection very simple. This thesis shows how these combined features can reduce modeling cycle-time dramatically in the context of air defense planning. Also, and of great interest to small nations, the high level of abstraction and reusability achieved by the component architecture may allow the functions of domain expert and simulation analyst to be combined in one individual.

x

ACKNOWLEDGEMENTS

I. INTRODUCTION

I want the future now
I want to hold it in my hand
I want the Promised Land

Peter Hammill "The Future Now"

In the 1991 Gulf War the United States Air Force demonstrated just how quickly a modern Integrated Air Defense System (IADS) could be destroyed. Stealth and precision guided munitions (PGM) were keys to the swift success. Offensive air power seemed to have had the upper hand at this point. Having a clear picture of what the enemy is doing is the alpha and omega in war. This is exactly the purpose for which sensors in their many variations are collecting data. Without sensors one is left blind, just as the Iraqi Air Force in Operation Desert Storm was. Stealth and PGM certainly presents a challenge to air defense systems, but solutions to this challenge are under development. Research into sensors for air defense cover technologies from bistatic radar to space based systems and infrared search and track systems (IRST). For an example, the U.S Marine Corps Science and Technology Program Plan for fiscal Year 1998 includes a research project titled "Advanced Targeting Sensor Technology Program", concerning the use of passive sensors in air defense.   What is the effect of stealth? What effect can IRST have when integrated into an air defense system? What mix of sensors is best in current and future scenarios? This problem area is not entirely new. Sensors have played an important role throughout the history of air defense.

A. THE ROLE OF SENSORS IN AIR DEFENSE

Although many factors contribute to the success of any military operation, it has long been recognized that information is one of the most important-information in many different forms and acquired on many different time scales.

Technology for the United States Navy and Marine Corps 2000-2035, Becoming a 21st-Century Force

Success can be dangerous. In 1991 the US Air Force put the Iraqi Air Defense System out of action with stunning effectiveness. In 1940 Herrmann Göring volunteered the Luftwaffe to defeat the British Fighter Command after the German successes in France. Just before the Battle of Britain started he sent the following message to his airmen: "From Reichsmarsschall Göring to all units of Luftflotte 2,3 and 5. Operation Adler. Within a few days you will wipe the British Air Force from the sky, Heil Hitler" (Terraine, 1988), only to get his nose severely bloodied. In fact, the Luftwaffe never truly recovered from the losses it took in the Battle of Britain. Herrmann Göring and the rest of the Luftwaffe high command held a strong belief in offensive air power. But they were wrong. This attitude seems to have taken hold after Operation Desert Storm. For an example, in his book "The Future of War" George Friedman claims that the United States Armed Forces will be able to dominate warfare well into the next century through the use of modern long-range precision weapons (Friedman, 1996). He may be wrong too. Technology is often available to those who crave it. If Friedman is right, too many players have too much to lose to remain indifferent. Technology in itself favors neither the offensive nor the defensive in the long run (Creveld, 1989). In the Battle of Britain it was precisely the use of a new type of sensor that wreaked havoc on established beliefs. There is reason to believe that that may be the case again.

Air warfare is by now such a well-established and important part of warfare that it is easy to forget just how young it is. Air warfare proper is only about 60 years old, whereas land and sea warfare have traditions measured in hundreds or even thousands of

years. Theory about air warfare had already appeared in the beginning of this century, just after the first flights of the Wright brothers. Notable theoreticians included the American General William Mitchell, and the infamous Italian General Giulio Douhet. Would it be possible to defend against fleets of bombers? Without experience, the theories tended towards the extreme. For example, Douhet believed that nothing could stop the bomber. He pushed his message so incessantly that the Italian high command court-martialed him and put him in jail for a while to cool off. He was later released and promoted to general when the course of events temporarily proved him to be correct. Billy Mitchell, also quite outspoken, took on the American military leadership and was court-martialed as well. The Second World War was to prove both men both right and wrong.

The Battle of Britain was the first large scale battle waged by Air Forces only. A definite success for the British Integrated Air Defense System (IADS), "Chain Home," was the first large scale integrated military system in which detailed flow of information and central control was used to maximize the effectiveness and efficiency of the system (Creveld, 1989). The success of Chain Home was largely due to the development of a new sensor at the time, radar, favoring the defender. However, there were other important features of the system. Chain Home's cofounder and operational top leader, Sir Hugh Dowding, had a good understanding of the use and integration of modern technology into military systems. He developed and nurtured a good relationship with the scientists working to build the system. Most importantly, the emerging technologies were used to integrate multiple information sources, communication and weapons into a tailor-made military system. The system allocated resources, both to meet the need of the moment, but also with long range considerations in mind. It was, in short, the first large-scale scientifically managed defense system. Optimistic at first, the Germans launched "Adler Tag," their code name for the main opening attack in the effort against Fighter Command. After a protracted attrition battle, the Germans gave up and turned their efforts eastwards towards the Soviet Union. Analysis has shown that the Germans

3

failed to understand the nature of the British air defense system, mostly judging it by the technological status of the radar system. Although the radar employed by the British was crude by German standards, they were only one part in a much more elaborate structure (Terraine, 1986). When classified records were released after the war it was clear that the British also utilized radio (a passive sensor) to listen and triangulate, which gave them an early-warning about German attacks.

The different sensors were used to build up what today is called the "Recognized Air Picture" (RAP), which is a "map" of all enemy and friendly activity in the airspace. The RAP was important in more than one way. In addition to showing just how and when its own resources should be allocated to defend critical resources, it gave Fighter Command a possibility to rest and recuperate between attacks. Armed with an early warning of German attacks, often hours in advance of the first German aircraft reaching the English coast, the British could prepare their defenses for the onslaught. The integration of multiple sources of information into a command and control system let Sir Hugh Dowding manage his relatively meager resources nearly optimally. He was able to inflict higher attrition on the Germans than they were willing to absorb. However, Chain Home had its weak spots, most notably the radar stations. The Luftwaffe failed to attack the weak and critical spots in Chain Home vigorously enough. When the Luftwaffe turned to bombing cities, this gave the British IADS relief from the threat of being blinded. Dowdings primary objective was to stay alive as a credible threat to the Germans, always defending the most critical resources, but never using too much to be left empty-handed the next day. He was forced to protect the infrastructure of his own air defense system and at the same time protect vital industries. To this very day, an enemy's IADS ranks among the top priority centers of gravity, and in that system it is the sensor system that has traditionally been the most critical and vulnerable part. In the aftermath, historians have marveled over the incredible management of Chain Home, emerging as it did victorious from a task that at first seemed impossible. Sir Hugh Dowding was among the first military leaders to have an intuitive understanding for the

application of science to the management of military forces in war. He was perhaps one of the first modern military leaders to apply Operations Research to his conduct of war, even before the science had been firmly rooted as an aspect of modern war-fighting. Clearly, Dowding fully understood the opening citation of this paper, and utilized it to its full potential.

Since the Second World War air warfare and air defense has become tantamount to warfighting. Most analysts will agree that air superiority is crucial to success in modern war. There are many interesting case studies of air warfare following the Second World War. The Arab-Israeli wars in 1967, 1973 and 1982 all include air warfare and air defense as pivotal elements (Cooling 1994). In 1967 the Israelis blasted the Egyptian air defense system out of commission in about one day of fighting, only to see it rebound with terrible revenge in 1973 when the Egyptian forces had learned their lessons and dealt the Israeli air force a serious blow. The Egyptians had to move their ground forces out of their own air cover because of a strategic error committed by Syria. This gave the Israeli Air Force time to rebound. In the Bekaa valley in 1982 the Israelis again showed the world how quickly a Soviet-style air defense system could be decapitated when they carried out a swift and enormously well-coordinated attack on the Syrian air defense system. The American Armed Forces studied the campaign and surely got first-hand knowledge about the precise tactics used by the Israeli Air Force. This knowledge would be put to excellent use ten years later in Iraq.

From the Second World War through Operation Desert Storm, offense and defense each had their day in the sun as new technologies were introduced, tactics developed, used, and eventually successfully countered. Neither side had a clear, long lasting upper hand. In Operation Desert Storm the coalition forces swiftly and effectively paralyzed the Iraqi air defense system, giving the most recent round to the offense. Although the Iraqi air defense system was modern by the standard of the day,

5

and was very large, its sensor system was almost solely based on active radar (Hallion, 1992). As we have seen, this was the critical and vulnerable part of the system.

The Norwegian Advanced Surface-to-Air Missile System (NASAMS) is a newly developed SAM system that meets many of the challenges created by the new offensive technology. NASAMS is a unique system consisting of air defense components that are put together in a network. A typical NASAMS unit consists of a three-dimensional pencil-beam radar, a Fire Distribution Center (FDC), and three six-missile launchers. The missile used is the AIM-120 AMRAAM, originally intended for use on interceptor aircraft. The FDC is manufactured in Norway from commercial off-the-shelf components. Up to four FDC's can be networked to share targeting data and control any launcher. This assemblage of air defense components achieves a combination of high firepower and survivability. Still, NASAMS depends on active radar. The increasing sophistication of precision guided munitions such as High-Speed Anti-Radiation Missiles (HARM) is an identified Achilles heel of the system. Great interest has therefore been placed on passive sensor system components.

B.   SIMULATION AND AIR DEFENSE PLANNING

Just what will be the impact of new sensors, and how should they best be utilized? This is among the most central questions that the air defense analyst must be asking. The analysis must try to gain insight into the effect of different sensors and configurations in relation to many different modern threats. The analyst faces a combination of uncertain data and very complex system behavior. Of all modeling tools available, system simulation is perhaps the only one capable of capturing the behavior of such systems. Simulation modeling is the most widely used technique for analysis of military problems for this very reason. Ideally military simulation modeling should be quick, cheap, and yield precise answers. However, many current models are large, monolithic and hard to understand (DMSO, 1995). They can seldom be used in a flexible manner to study problems that are related to, but yet slightly different from what

the model was intended to do in the first place. Building and using a simulation model is an expensive, slow and cumbersome activity. This leads to low productivity. We are concerned with the efficiency of simulation modeling, meaning that combination of timeliness, correctness, flexibility and affordability that support military decision-making. Unfortunately, simulation cannot currently support high-tempo operational decision-making.

Reuse is the key to increasing effectiveness in simulation modeling. We consider reuse on two levels. On one hand the abstractions that make up the model, and on the other, the corresponding implementations. We shall have to reuse both abstractions and implementations extensively to gain efficiency. Research has been conducted on Model Management, a term that was coined in the mid 1970's in the context of work on decision support systems. The seminal work in the area of Model Management and Model Integration was done by Geoffrion (1987), and a more recent overview of this area is given by Krishnan (1997). Structured Modeling was developed as a comprehensive response to perceived shortcomings of modeling systems in the 1980's. It is a systematic way of thinking about models and their implementations, based on the idea that every model can be viewed as a collection of distinct elements, each of which has a definition that is either primitive or based on the definition of other elements in the model. There has been much theoretical work on the construction and use of models. Davis (1993) treats the problem of variable resolution modeling and cross resolution model connection. Blanning (1991) provides a volume of articles spanning from theories on models and model integration to object-oriented approaches to model management. Zeigler (1976) has written extensively on theories of modeling and simulation and related issues. Taken together the above mentioned work puts up a very comprehensive and interesting theoretical framework for modeling, especially for classifying and understanding the different abstractions.

We must also be able to effectively implement our model abstractions. Since programming is the task of mechanizing the model abstractions into code, we must be concerned with software engineering and programming. To our knowledge the work on Model Management and Model Integration has not produced any practical application or methods that have gotten widespread use or attention. Whereas it is highly interesting theoretically, we have chosen to approach the problem of simulation productivity and efficiency in a more empirical fashion. Object-Oriented Software Engineering (OOSE) is a field that concerns itself with effective software engineering, with special focus on reuse (Jacobson, 1993). A central aspect of OOSE is the notion of a use case. Use cases are specifications of courses of events that form the interaction between the system and its users. However, the uncertainty and unpredictability involved in military planning situations effectively preclude precise definitions of use cases before the situation occurs. Consequently, it is highly desirable to be able to rapidly tailor the software to the situation at hand. Component Software is a technology that addresses this problem. We now go on to discuss the prototype software component architecture developed in this thesis.

C.   MODKIT - A JAVA COMPONENT ARCHITECTURE

The author has conducted a series of software experiments to gain insight into software component technology for simulation modeling. The resulting component architecture is implemented in the object-oriented programming language Java, and has been named Modkit, short for Modeling Kit. From this point on Modkit will refer to both the software architecture and the small library of components that has been developed for the sake of the simulation models used in this thesis. It should be clear from the context whether we are referring to the architecture or the components. Modkit is the result of a series of software experiments, encompassing the use of several different software patterns such as the Mediator, Composite, Decorator, Chain of Responsibility and Command patterns (Gamma, 1997). The first set of experiments

8

focused on basic composition and event handling. The most important lesson from this experiment was that events should be passed through globally standardized interfaces. The second experiment concerned discrete-event simulation of queuing networks using components. This experiment yielded the very important result that interactions between components can most beneficially be handled by a third component, a mediator. Finally and most importantly for the thesis, most of the work has gone into creating a software component architecture for discrete-event simulation. A late yet very important lesson learned was that access to component data must be possible through a simple interface. Modkit is comparable to Sun Microsystem's Java Beans™ in some respects. Java Beans are mostly used for creating graphical user interfaces for programs, and this has unfortunately limited its usefulness. Modkit has been specifically developed to be more general than Java Beans. Finally Modkit has been developed in the context of building models for OR purposes.

The rest of this thesis goes on to discuss what software components are, followed by a tour through the important features of the Modkit software component architecture. Following, this the Modkit component library is introduced and demonstrated. With this background we move into a study of the impact of IRST systems on the effectiveness of integrated air defense systems. After a discussion of the results of this modeling effort we summarize the experiences and point to some of the very important benefits that have accrued from the component approach.

II. SOFTWARE COMPONENTS FOR SIMULATION

One thing can be stated with certainty: components are for composition. Nomen est omen. Composition enables prefabricated "things" to be reused by rearranging them in ever new composites. Beyond that trivial observation, much is unclear.　　　—

Clemens Szyperski

Can simulation modeling benefit from the component paradigm much as other industries have? Most experts think the answer is "yes" for software development in general (Jacobson, 1993), (Szyperski, 1997). Professor Niklaus Wirth, one of the world's leading computer scientist, and the father of Pascal, has recently developed Component Pascal. In some areas, like building graphical user interfaces for programs, software components like Java Beans are getting a solid foothold, and a market is being created. The benefits of using loosely coupled software components for military planning purposes are discussed by Bradley and Buss (1997). Finally, the philosophy underlying Modkit closely resembles the architecture of connected components so successfully used by the NASAMS system.

The key issue addressed by software component technology is productivity. The productivity of simulation modeling is intimately linked to the productivity of software development. Not using component technology has the following known drawbacks (Szyperski, 1997)

- Requires reinvention of solutions

- Strict limits of growth and reusability

- Uneven quality and suboptimal solutions

The combination of diminishing military budgets and rapidly developing technology demands effective and efficient analysis tools. Most current models cannot

11

provide what the air defense planner needs. The productivity of simulation modeling need be increased dramatically. Software component technologies are perhaps the most promising solution to this need.

Imagine the situation if simulation models could be assembled from a library of generic components. This would be akin to how the electronics industry assemble products from standardized components. Instead of inventing the same functionality again and again, the engineer can look for components with the specific capability. Rather than thinking about how to implement specific aspects of an entity in a model by programming, the analyst could be free to think about the system he is trying to model: what entities does it consist of, and how do the entities interact with each other to produce the systems behavior? Furthermore, the analyst could use a simple model as a baseline to explore in which direction further effort should be expended. The analyst would in effect do modeling by stepwise refinement. This would be a natural way for an analyst to work, and very desirable from an efficiency point of view. However, this approach is not possible with existing software architectures. This is one of the main reasons why simulation modeling is such a complex and time-consuming process. A model consisting of loosely coupled interacting components can meet this challenge. To support analysis for air defense planning, the software architecture used should facilitate reuse, integration and extension of a growing library of model components.

Using component technologies has the following known benefits (Szyperski, 1997)

- Components improve in quality much faster than "hand-crafted" solutions

- Growth can become less limited

- The user can benefit from combined productivity and innovation of all component vendors

12

- Large potential for reuse

- High quality in each component

Modkit is implemented in Java to support modular discrete-event simulation for the air defense problem in this thesis. While a full-scale simulation analysis of the role of modern sensors in air defense is not possible within the scope of a masters thesis, we can nevertheless carry out the first exploratory stages of such work. The component architecture should make it easy to grow the model to higher levels of fidelity and complexity.

A. WHAT IS A SOFTWARE COMPONENT

Many different definitions of software components exist. Szyperski (1997) provides a very good overview of the state of the art in software component technology. The Software Engineering Institute, a U.S. Department of Defense sponsored research center, has published a study on component-based software engineering where many of the terms and concepts are explained and discussed (Brown, 1996). Jacobson (1993) provides an insightful chapter on software components in his book on software engineering. This thesis focuses on the specific context of air defense planning and simulation. The experience from developing Modkit has led to the following list of lessons learned:

- Components should be able to work stand alone as separate entities

- Components should communicate by passing messages (push-model)

- Components should be able to provide data to each other (pull-model)

- Components must be composable, meaning that several components are connected together to form a system of components.

- It should be possible to compose components recursively

- The components must be loosely coupled.

- Connecting components should be simple

- Components must be of high quality

- Component documentation should be very good

It is interesting to note that all of these lessons learned with the exception of recursive composition can be found in the literature on software components. This indicates that our conclusions may be fairly context-independent.

Several analogies have been used to enhance the understanding of components. To be sure, understanding something new by invoking a suitable analogy from a known area is helpful, but some care is required. Some of the notions used include:

- Lego Blocks

- Integrated Circuits

- Stereo Equipment

- The Personal Computer

- Mechanical Engineering

These analogies may aid our intuition of what a software component may be, but unfortunately, the analogies fails to capture some of the most important (and unique) aspects of software components. First of all, we can tailor software components to a large degree. Secondly, software components can be recursively nested. Thirdly, software components can be instantiated any number of times.

14

We should think of a software component as a factory for components, rather than a fixed product. Software components are unique. We cannot expect to be able to produce software components from analogy to other fields. Creating software components is a separate engineering activity (Szyperski, 1997). However, it is still useful to invoke analogies, as we will, when discussing this new technology. We now turn to the practical issues of component documentation.

B. FEATURES OF COMPONENTS

Software components should be documented for ease of use by both end user and component programmers. Components must therefore be documented from several viewpoints. We have provided the component fact-sheets used in this thesis as a suggestion of how components could be documented. First of all, one should focus on understandability (Jacobson, 1993). A component represents an abstraction, and we should use any means available to convey this abstraction to the component user. We have chosen to focus the fact-sheets on the component user, letting the first page of the fact-sheet contain the most highly abstracted information. More detailed data, like implementation code, comes towards the end. The method used is closely modeled on the way the electronics industry provides fact-sheets for their components.

The fact-sheet also fully reflects the important attributes of a component. Hence the following discussion also serves as an introduction to the important aspects of the Modkit component architecture. The items are discussed roughly in the same order as they can be found in the fact-sheets.

1. Focus - the Functional Aspects

The simulation expert using the components will be most interested in the functional aspects of the component. What does the component model? This could range from a generic component for modeling motion, to a more complex model of an aircraft. Therefore, the first part of the fact-sheet should provide drawings and graphics

15

that give the user a good overview of what the component does and how it can be connected to other components. Ideally the user should not have to go beyond this first page to start using the component. No formalism should stand in the way in presenting the first view of the component.

2. Syntactic and Semantic Aspects

When components are connected we must assure that they are compatible. That is, they must share a common standard for what happens on the connections. In an electronic setting this would correspond to electrical characteristics such as voltages and frequencies. But this is only the very basic level; there must also be a standard for the signals being passed such that the components can interact in a sensible fashion. The situation is similar for software components. To be connectable the components have to be "line" compatible with each other. This happens on the syntactical level, and is effectuated in Modkit with Java interfaces. We can call this syntactic level of standardization the "wiring level." The next level of standardization is the so-called semantic level. Here we must ensure that the components can "make sense" out of the messages passed between them. There is no mechanism to formally enforce semantic level standards, and we make no attempt at building this level of intelligence into the library components. Rather, we let the component user be responsible for connecting components such that the components interact sensibly. However, a section in the component fact-sheet is dedicated to the semantic interface of each component. In addition to the division into a syntactic and a semantic level we must also distinguish between what the component provides to the outside world (outgoing) and what the component can handle (incoming), both in terms of messages and data (Szyperski, 1997).

Table 1 summarizes what is provided in the documentation for component interfaces. Note that the syntactic interfaces are, and should be, the same for all

16

components. The syntactic interfaces can thus be taken for granted for all components in Modkit.

	Syntactic(Interfaces)	Semantic (Classes)
Outgoing	ModPropertyProvider ModEventSource ModEvent	Generated Events Provided properties
Incoming	ModPropertyUser ModEventListener ModEvent	Handled Events Used properties

Table 1. Component Interfaces

3. Syntactic Standards – Java Interfaces

The important aspects of components in Modkit are interrelated. But perhaps the most important feature in order to produce a component standard is loose coupling. In Modkit this is achieved by a combination of Java interfaces, late binding, and dispatcher classes. Combined with Java's introspection capability this achieves a high degree of loose coupling. The syntactic level of standardization ensures that all components can pass messages and data to each other in a simple and straightforward fashion. The Java interface mechanism is perfect for this use. We ensure loose coupling by defining five interfaces. We have defined three interfaces to handle events, and two interfaces to handle properties (data). Each interface consists of very few abstract methods. Figure 1 uses the "Integrated Circuit" analogy to give an intuitive explanation of the interfaces.
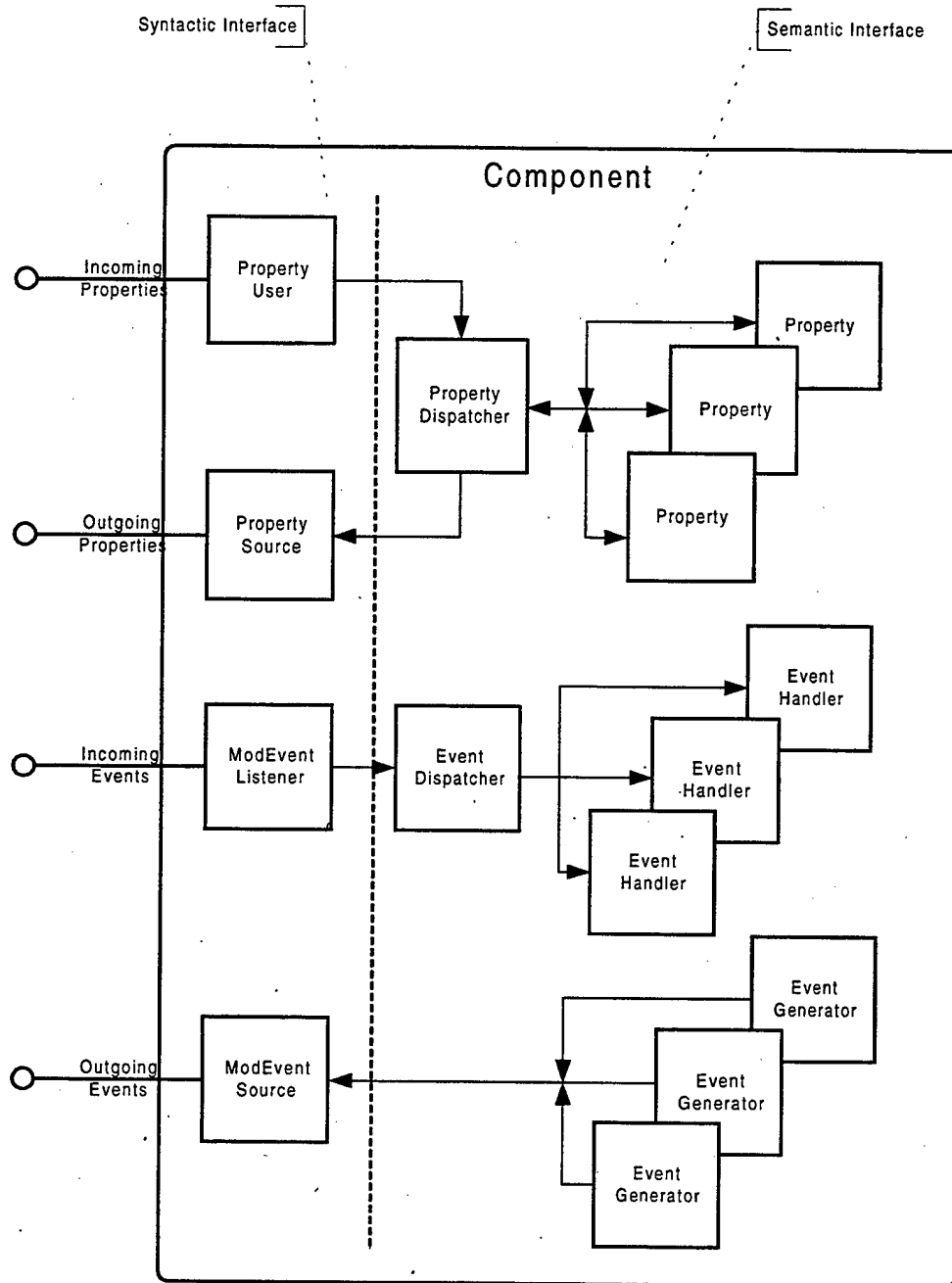
Figure 1. Software Component

    The syntactic interfaces on the left side of the dotted line correspond to standardized connectors or pins that can be used to connect the component as indicated on the drawing. The functional blocks on the right hand side correspond to the semantic

18

interfaces. All functional blocks in the picture of the component correspond one-to-one to the Java source-code for the component interface definitions and implementations.

4. Semantic standards – Events and Properties

a. *Messages as Events – the Push Mechanism*

When a component transmits (or "fires") an event it signals that something potentially interesting to the outside world has happened inside the component. The content of the message is defined in the event object transmitted. The event itself is an instance of a class that implements the event interface. The event passing mechanism is a "push" mechanism in the sense that the component fires the event, not caring how its listeners react to that event (Szyperski, 1997). Modkit components are loosely coupled with respect to message passing. In effect, an event is just pushed out to the rest of the system, which can then decide how to respond. The component fact-sheet lists the class names of the event classes, and a corresponding description of what condition has triggered this event. Comparing this with how other components handle the same event, the component user can decide how components are going to interact with respect to events.

b. *Data as Properties – The Pull Mechanism*

Sending and receiving messages is a necessary condition for loose coupling, but it is not sufficient in itself. By their very nature, components must limit what they can do. In the design of a component it must be possible to assume that services can be provided by other components. Said another way; sometimes data is needed inside a component that must be found outside of the component. In keeping with Java Beans terminology we call data "properties." A property is a piece of data that a component has, uses, or can provide.

19

The PropertySource and PropertyUser interface defines a unified way to treat properties across all components. Whereas the Java Beans standard uses accessor methods on a property-by-property basis, Modkit defines a generalized accessor method for all properties. Therefore, Modkit components are loosely coupled both in a push and a pull fashion. These are the necessary and sufficient mechanisms to function in a collection of components forming a system. Properties are documented in a similar way to events. The component user can find the name of the properties, their class names, and a description of what the properties are in the component fact-sheets. Szyperski (1997) has an excellent discussion on push/pull model of programming.

5. Composition

How can components be put together to provide a composite model or system? Ultimately, building models should involve assembling ready-made software components rather than writing individual lines of code. This activity is also programming, but at a higher level of granularity (Szyperski, 1997). Composition is a much different method than object-oriented software development. Traditional software development is a top down process resembling a functional decomposition of a system. Object-oriented development is both a top down and a bottom up process. However, assembling components is a pure bottom up process. The latter method is fast and natural, but requires the existence of a library of mutually compatible components. Given a set of useful components, new systems can be created by composition. The development process is mainly concerned with how components are grouped together and connected to arrive at a new system. If a suitable component is lacking at any stage, the user should be able to turn to a component programmer, or perhaps act the programmer, to produce the desired component. In that case Modkit allows the component programmer to benefit fully from all the object-oriented features of Java. Specifically, code in existing components can be reused via implementation inheritance. There will be many different ways of composing the components, and it will not in

general be possible to anticipate all possible compositions. Therefore, a small example of possible compositional schemes are shown, and the rest left to the creativity and skills of the component user. The purpose of the interfaces and standards defined in Modkit is to make composition possible with minimum effort. This feature of Modkit brings us closer to truly reusable software components than the object-oriented programming model alone. The syntactic interface of a Modkit component remains constant under composition. That is, viewed from the outside, the composite is syntactically just like any other atomic component. The semantic interface of a composite is the union of the semantic interfaces of the constituent components. This allows us to recursively nest components, and to handle complexity by a divide and conquer method. Also, this feature makes it possible to handle component interactions in a simple and standardized way. The demonstrations in the next chapter will show this in practice. Since our component library is small we have only very few examples to draw from, but we will provide examples of both atomic and composite components in action.

It is interesting at this point to note that building models from components finds an abstract parallel in Structured Modeling (SM). In SM atomic models are called primitive. Modkit comes very close to implementing in practice the abstract view of a model in SM. SM was developed specifically for modeling as practiced in OR/MS. Geoffrion (1996) mentions in his article that "likely topics for further work include discrete-event simulation." Our starting point was discrete-event simulation for air defense analysis. Models in SM are thought of as consisting of a collection of distinctive elements, each of these elements being either primitive or composite, and the elements are organized hierarchically as a rooted tree. The dependencies among the elements are diagrammed as a directed acyclic graph. This so-called dependency graph can be computationally active. We can build composites using Modkit just as in Structured Modeling, but we are not limited to the tree structure. Components can be connected in a fashion resembling entities in SM, but we have chosen our own name for

the computationally active arc, namely the mediator, and we have chosen to think of the connection as a bi-directional interaction between entities. (Geoffrion, 1996)

6. Interaction

A good example of an interaction between components is a sensor detecting a moving target. Although both entities take part in the interaction, it would defeat the purpose of the component architecture to model the interaction in either of them. reason for this is simple. Doing so would require that every component included code for interaction with all other components, and each component's implementation we grow enormously. Furthermore, existing components would have to be rewritten to account for new additions in the component library. This is precisely the messy situation we want to avoid. Interactions between model entities (components) are the dominant aspect of air defense simulation models. This is just a special instance of a problem that occurs in creating a software component, namely, where to draw the boundaries of the functionality of the component. Quite clearly one cannot ask of a component that it "do everything." Certain assumptions about the availability of existing components must be made. For example, the speakers in a stereo system assumes the existence of an amplifier. The answer to this problem in Modkit is to use the mediator pattern to handle component interactions. Each mediator is designed to take care of one and only one type of interaction between one and only one pair of components. The mediator corresponds to the computationally active arc in Structured Modeling, and it is also a recurring pattern in software engineering (Gamma 1995, pp. 273). With respect to the latter it is instrumental for reuse. The mediator pattern promotes loose coupling by keeping objects from referring to each other explicitly, and lets us vary their interactions independently. The mediator is syntactically just another component. This is extremely important since it allows us to connect components with different mediators over the course of a simulation run. As an example, a system including a sensor may detect a moving target. This interaction is taken care of by a specific mediator (we will

demonstrate this in the next chapter). But some time later the system may fire a missile at the moving target. The interaction between the moving target and the missile is of a different nature, and is handled by another mediator. This is possible because the syntactic interface is constant under composition. Compare this to Sun's component standard, Java Beans, which uses so-called adapters to glue together components with disparate syntactic interfaces. The Java Beans approach effectively precludes the dynamic composition achieved with Modkit. In Modkit, complex behavior and interactions are decomposed into small pieces that are simple to handle individually. As a side benefit, the amount of computer code in memory is limited to what is "going on" at all times, rather than carrying around "dead code". It should also be re-emphasized that the semantic interface of a composite is the union of the semantic interfaces of its constituent components. This is important because a given mediator can broker interaction between atomic or composite components so long as it is compatible with at least one component in each composite in the interacting pair. An example of this is the MovingSensor component in the demonstration in the next chapter. The mediator listens to the events from both of its components and can request data (properties), as it deems necessary. This ability is guaranteed by the constancy of the syntactic interface standard. Connecting the mediator to components is kept very simple by the same interfaces. No adapters are needed between the components and the mediator. The mediator decides the result of the interaction, and plays a key role for reuse by allowing components to become stable software entities. Perhaps most importantly, the mediator makes it easy to scale up a model. The fact-sheets contain a section for interaction that shows what mediators are compatible with the component.

7.   Source code

Finally the documentation provides the source code of the component. It may be necessary to understand the internal workings of a component to use it properly. Furthermore, a component is a factory for producing many components. We can make

instances of the component, each with different parameters. More importantly, the component programmer/user can benefit fully from Java's object orientation, since he can write new components by extending and thereby also reusing the implementation of a specific component. So we suggest that the full source code should be part of the documentation.

C.   SUMMARY OF COMPONENT FEATURES

We started this chapter by pointing out how efficiency in simulation modeling is closely related to reuse. Both the abstract aspects and implementations of model components should be reusable. We have introduced the important features of Modkit components through a discussion of how the components are documented, and we have shown how the fact-sheets support reuse of both abstractions and implementations. Although the documentation methodology suggested here is purely practical, many of the abstractions have much in common with those found in Structured Modeling. Loose coupling is achieved in Modkit by using Java interfaces together with properties and events. This assures component reuse by making composition possible. Component code reuse is supported by Java's implementation inheritance. Therefore we can reuse the component implementation on two levels, first as a component in a composite, secondly as a starting point for developing new components. Abstraction reuse is a function of the semantic interface standard, and corresponding documentation in fact-sheets. Components are interesting because they allow us to reuse all these important parts going into a simulation model. In short, we have packaged both our abstractions and implementations for easy reuse. This fact together with the scalability achieved by the mediator mechanism should increase simulation modeling productivity by a very large factor.

III. COMPONENT LIBRARY

To build a simulation from components we need a library of components. This chapter introduces the three components that make up the baseline library from which the air defense simulation is built. We give two demonstrations of how the components can be used as a precursor to understanding how the air defense simulation is constructed.

Motion through space and sensing of moving targets are the most important aspects in an air defense simulation. The two main components are for motion and sensing, with the third being a mediator for the interaction between a sensor and a moving target. The data sheets including the source code of the library components are given in the Appendix E.

A. MODELING MOTION

Spatial relationships and movement are fundamental to many military simulation models, but do not directly relate to the Modkit component architecture. For that reason a separate Java package has been devoted to vectors and the most used operations of analytic geometry. Following good object-oriented design practice, this class is built on an interface. Vectors can be of any length, that is, they can have any number of coordinates. However, it has been found most beneficial to use a four dimensional vector class named Coor4D. The RouteMover, Modkit's component for movement, makes extensive use of the Coor4D class and its operations. Loosely speaking, the RouteMover "implements" the Moving semantic interface. (We remind the reader again that there is no Java interface for this function, it exists only as part of our fact-sheet for the component. The use of the word "implements" should not be confused with the formal Java issue of implementing a syntactic interface.) It simulates moving from point to point in three-dimensional space. Between waypoints movement occurs with constant velocity. Among the properties of this component are the Route4D. The Route4D is a

stack of four-dimensional points. The RouteMover proceeds from point to point in this stack, "popping" the next element as it arrives at a waypoint. The fourth coordinate in the Route4D object is the time at which we want the mover to be at the location. When the RouteMover arrives at a waypoint it fires two events, VelocityChangedEvent and ArrivalAtLocationEvent. Any other component interested in the status of the RouteMover can register as listeners and be notified whenever the RouteMover arrives at a new location, or changes velocity. Also, the current position, velocity vector and speed is available at all times as properties, since the RouteMover is a source of these properties. It uses linear interpolation between its starting point and destination point to calculate its current position. Current time is taken to be the time available from the simulation time-master. When the RouteMover reaches its final destination it fires the MoverStoppedEvent.

B.   MODELING SENSING

We have chosen to model the sensing aspects with a variation of the so-called cookie cutter sensor (CCS). Usually a CCS is a just a circular area. Targets are considered detected once they are physically inside the area. Our component is similar, but three-dimensional, a sphere. The CCS "implements" the Sensing interface, including the incoming events VelocityChangedEvent, ArrivalAtLocationEvent, MoverStoppedEvent and the NewSignatureEvent, and outgoing DetectionEvent and UnDetectionEvent. If the CCS receives a NewSignatureEvent it checks the position of the signature. If the target is inside of the detection volume it "detects" the target. It signals this by firing a DetectionEvent. The CCS then calculates when the target can first exit the detection-volume and schedules a check at that time in the future. It registers itself as a listener with the target. Should the target change velocity the CCS will recalculate the first possible exit-time. When the target exits the detection-volume the CCS signals this by firing an UnDetectionEvent. Notice that the CCS does not list any properties relating to its location or motion in space in its outgoing semantic

26

interface, since it has not been constructed to model motion internally. This is a practical example of drawing the boundaries for a component's functionality. It has been assumed that the CCS can be connected to a component for motion through space, much as a radar system is mounted on an aircraft. Should no such component be connected to the CCS, it just reverts to a default user selectable static position. The CCS works in conjunction with any mover, and does not care if its moving component behaves like a missile or a tank. In the demonstration that follows later we will show composites of sensors and movers moving through space and sensing each other. All we have to do to make a moving sensor is to put a component for movement and a CCS in a container. If we later build a specific component for moving, such as a surface-to-air missile, the CCS can be connected to this future component, and we would have a composite behaving like a surface-to-air missile with an internal sensor.

C.   MODELING INTERACTION BETWEEN MOVERS AND SENSORS

Finally, we need a mediator to broker interaction between our CCS and movers. The SensorMoverMediator (SSM) class is a component designed for this purpose. The SSM is registered as a listener with two and only two components. One of the components is considered to be the target, the other is taken to be the sensor. Both components must adhere to the Moving semantic interface, and the sensing component must adhere to the Sensing interface. The SSM requests the current location and velocity vectors of the target and sensor and calculates when the target can first enter the detection-volume of the sensor. The calculation is redone if any of the components fire a VelocityChangedEvent before this time. When the SSM determines that the target is entering the detection-volume it hands off the target's signature to the CCS, and waits for a DetectionEvent from the CCS regarding that target. The SSM then rests until the CCS fires the UnDetectionEvent for the target, when it resumes business as before.

D. DEMONSTRATIONS

This demonstration shows two simple simulations. The first involves atomic components; the second uses two composites. In the first demonstration a RouteMover has been set up to go from location (10,0,0) to (-10,0,0). A CCS has been located at the origin. The drawings have been annotated to show where and when detection and "undetections" occur, and should be fairly self-explanatory. In the second example both components are composites. Each composite consists of a RouteMover and a CCS. The drawings have been annotated to show the events occurring. Components and connections for the atomic demonstration are illustrated in Figure 2 in which lightning bolts indicate that both the event and property connections have been established. Figure 3 shows the resulting simulation sequence.

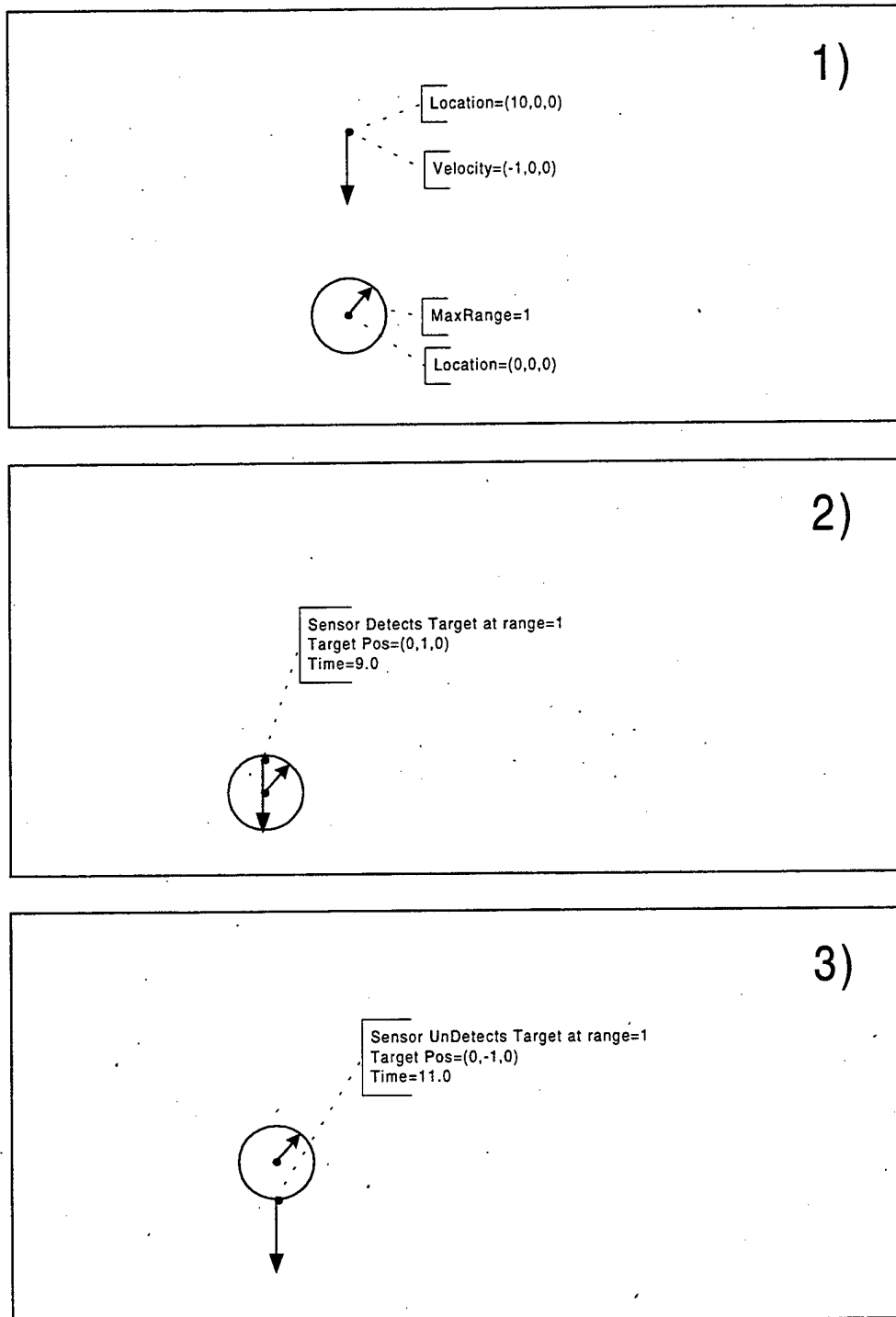**Figure 2. Components and Connections for Atomic Demonstration**

Figure 3. Atomic Demonstration

Figure 4 shows the composite components and their connections. Notice that two mediators are needed since both composites move and sense. One mediator is designed to consider one of its components as the moving target, and the other as a sensor. The fact that one composite detects the other is quite independent of what the other sensor is doing. In the composite demonstration a RouteMover and CCS has been put into a container to form a moving sensor. The mediation between two RouteMovers and a stand-alone RouteMover and CCS are identical. This is possible because the syntactic interfaces are constant under composition, whereas the semantic interfaces are the union of the individual interfaces. The mediator thus sees no difference between the atomic and composite components from the outside.
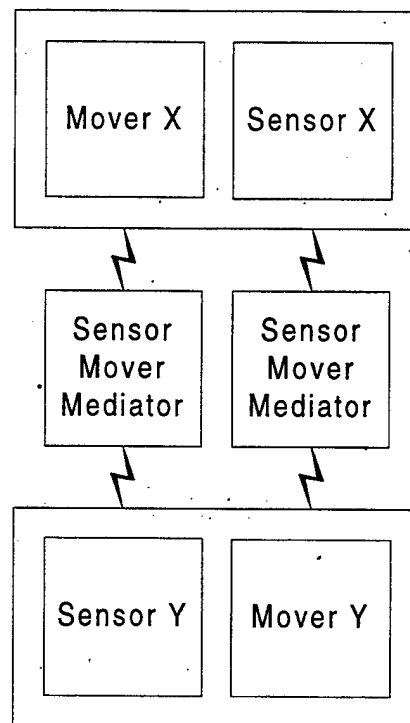
```
+---------------------------------+
|  +-----------+  +-----------+    |
|  |           |  |           |    |
|  | Mover X   |  | Sensor X  |    |
|  |           |  |           |    |
|  +-----------+  +-----------+    |
+---------------------------------+
         |              |
    +-----------+  +-----------+
    | Sensor    |  | Sensor    |
    | Mover     |  | Mover     |
    | Mediator  |  | Mediator  |
    +-----------+  +-----------+
         |              |
+---------------------------------+
|  +-----------+  +-----------+    |
|  |           |  |           |    |
|  | Sensor Y  |  | Mover Y   |    |
|  |           |  |           |    |
|  +-----------+  +-----------+    |
+---------------------------------+
```

Figure 4. Components and Connections for Composite Demo

Figure 5 shows the simulation sequence for the composite example. In the interest of space the illustration does not cover the undetection phase.
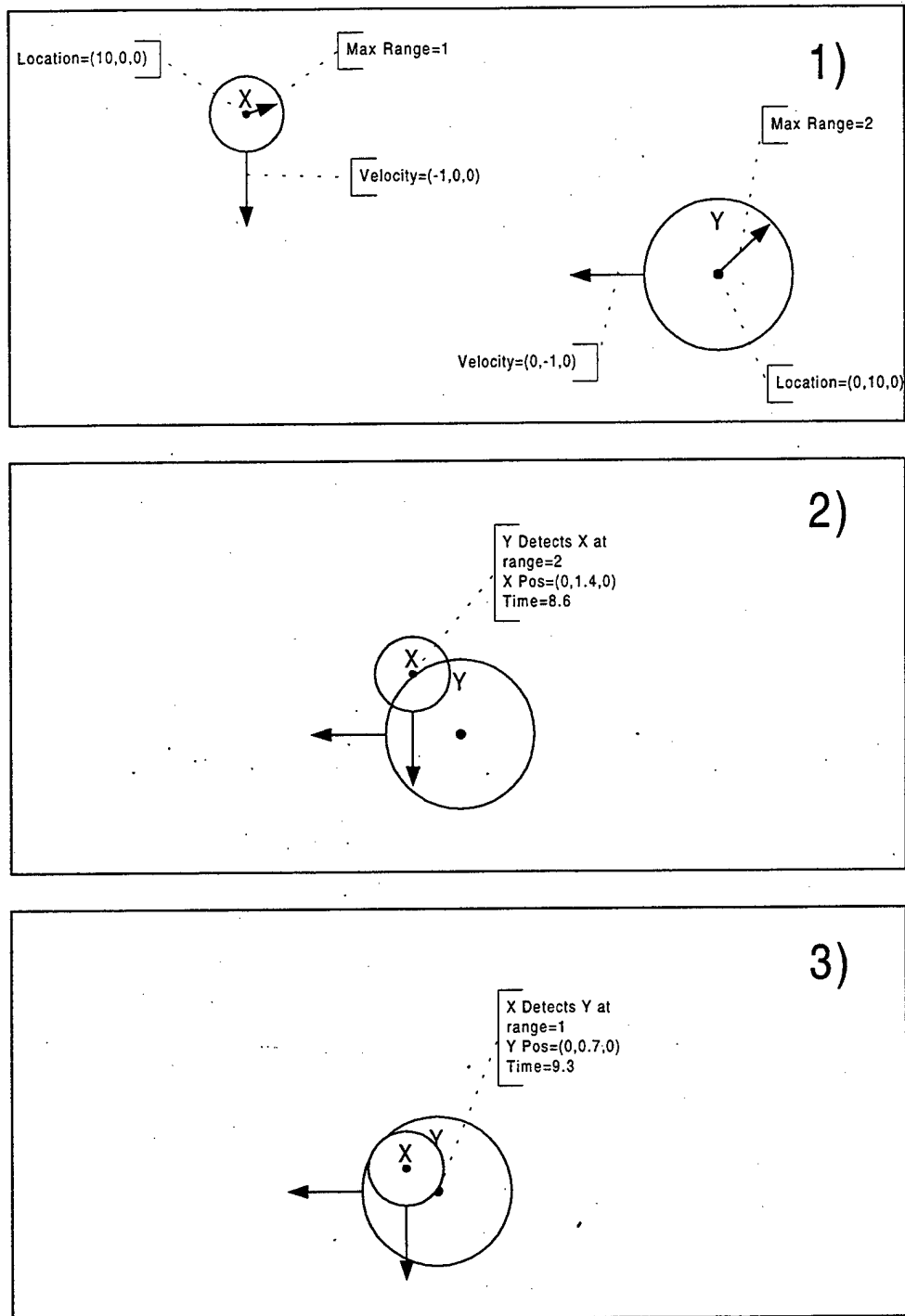
Figure 5. Composite Demonstration

31

In this chapter we have introduced and demonstrated the Modkit component library. Although consisting of only three components, moving, sensing and interactions between entities that move and sense are recurring aspects of air defense modeling. In the first demonstration we saw how two atomic components and one mediator operated to mimic a static sensor detecting a moving target. In the second demonstration we produced a moving sensor by connecting a mover to a sensor and putting the two components into a container, much like a radar can be mounted on an aircraft. Also, since any number of mediators can be connected between components, we were able to let both composites mutually detect each other. We now go on to show how the Modkit architecture and the small component library can be used to build a larger scale simulation model.

IV. IRST SYSTEMS AND AIR DEFENSE ENGAGEMENT OPPORTUNITY

> Now the sirens have a still more fatal weapon than their song,
> namely their silence. And though admittedly such a thing has never
> happened, still it is conceivable that someone might possibly have
> escaped from their singing, but from their silence certainly never.

Franz Kafka, The Silence of the Sirens

This chapter shows how the simulation components can be used in an exploratory analysis of the feasibility of using networked IRST sensors. Two models are developed. The first model gathers data on how the targets are tracked by the radar and IRST sensors as a function of detection ranges for both sensor systems. The second model extends the first model with components for simulating the flights of surface-to-air missiles. All parameters used in this study are from open sources and must be considered to be rough approximations. Furthermore, to making any but the most tentative conclusions would require access to classified data. Stealth is a collective term for reduction of signature and applies to all parts of the electromagnetic spectrum and sound (Knight, 1989). Stealth is achieved using a combination of techniques such as Radar Absorbing Materials (RAM) and specific airframe shaping. Stealth in the radar portion of the electromagnetic spectrum has been very successful. IR emissions from engines, exhaust, and aircraft surfaces warmed by skin friction are more difficult to reduce. To what degree can IRST sensors compensate for a short detection-range due to low target RCS? Precision Guided Munitions include cruise missiles and air delivered munitions such as laser guided glide bombs and the new Joint Standoff Weapon. Modern precision guided munitions such as Cruise Missiles and HARM have made fixed active sensors (radar) extremely vulnerable. In the late 90's there has been a trend towards providing air defense systems with passive rather than an active air defense alerting capability. This trend has been most notable in locale/mobile air defense units. However, nation-wide air defense networks are now vulnerable, as Operation Desert

Storm showed so vividly. Passive electro-optical sensors have limited range, and cannot individually act as substitutes for long range surveillance radar. What if the passive sensors were connected in a network? Fusing IR and electromagnetic sensors may make the system less susceptible to target countermeasures and destruction of one sensor by a preemptive strike.

A.   NASAMS AND IRST

NASAMS (Norwegian Advanced Surface-to-Air Missile System) is a modern Ground Based Air Defense system developed jointly by Norwegian and American industry. The system is now in operational use in Norway, and has proven its worth in several live firing exercises in the United States. A typical NASAAMS battery consists of

- 4 ARCS (Acquisition Radar and Control Center)

    - 1 FDC (Fire Distribution Center)

    - LASR (Low Altitude Surveillance Radar)

- 3 NTAS (Norwegian Tracking Adjunct System)

- 9 LCHR (AMRAAM launchers)

Up to four ARCS can be networked together to share data. All FDC's can provide targeting data to any other FDC in the system. Figure 6 shows a picture of the Low Altitude Surveillance Radar.
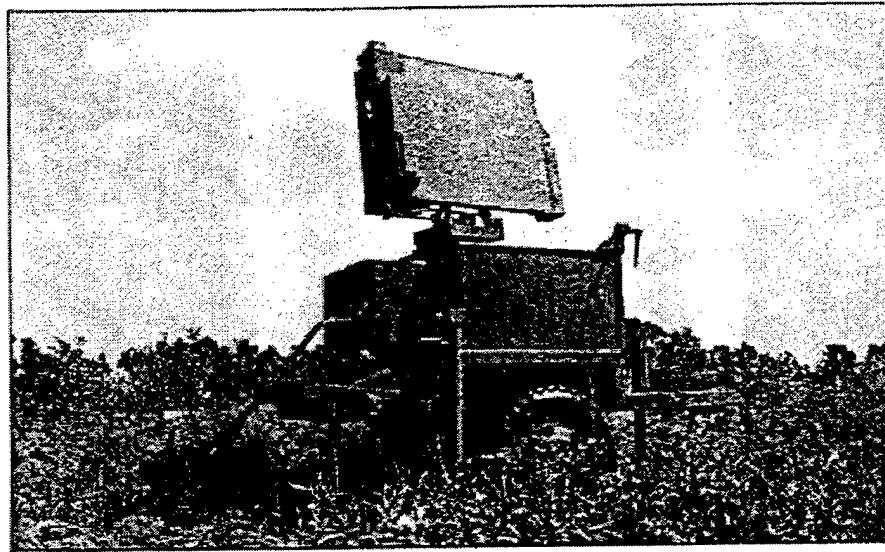
Figure 6. Low Altitude Surveillance Radar (LASR)

Sensors that can provide target data in a suitable format can be connected into the system. The FDC carries out all functions for airspace control and track management. This includes track correlation and Jam Strobe Triangulation, target identification, threat evaluation and weapon assignment. The FDC also contains the Battery and Launcher communications equipment and interfaces to the radar control computer. The AN/TPQ-36 radar is a phased-array pencil beam radar that can track up to 60 targets simultaneously, with a maximum detection range of 75 km. However, the range falls off to 40 km versus low RCR fighter sized target. Each FDC can control a number of launchers, which can be positioned up to 25 km away from the FDC. The launchers each hold 6 AIM-120 AMRAAM missiles. Initially these were unmodified, but to increase the range of the system a new program, the AMRAAM Rocket Motor Enhancement, is under way to significantly increase the weapon's target engagement range and altitude capabilities. Figure 7 shows a photo of the AMRAAM six-missile launcher.
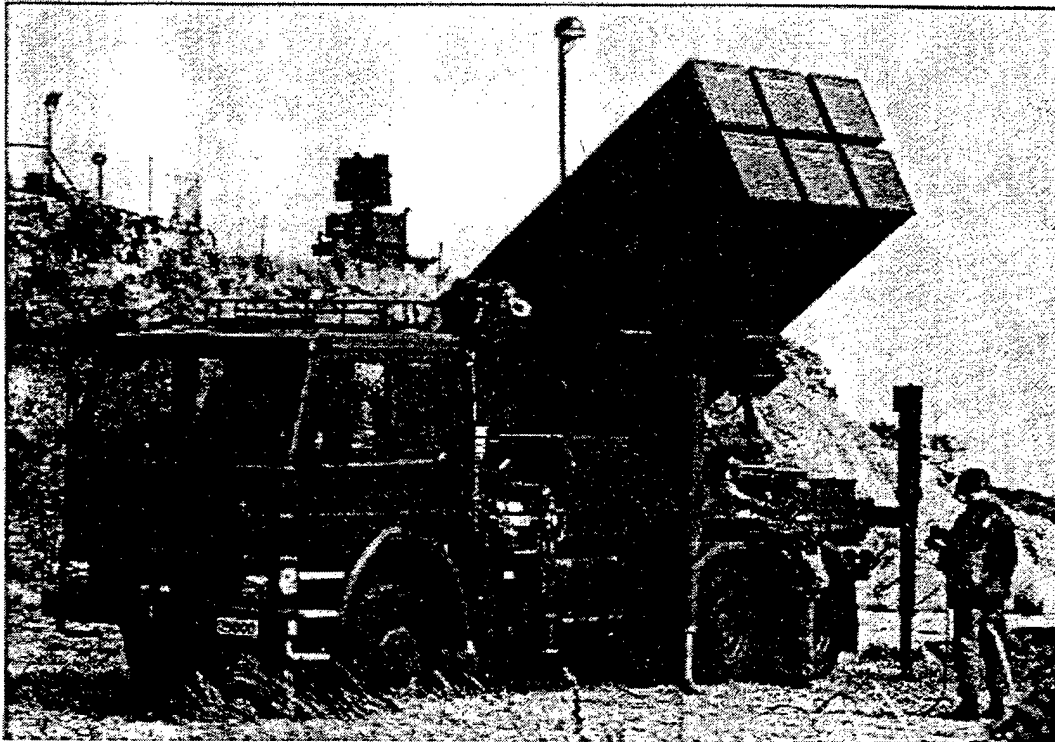
Figure 7. AMRAAM Launcher (LCHR)

Each FDC is also equipped with an IR sensor called the Norwegian Tracking Adjunct System (NTAS). It is used for visual target identification and raid size assessment in order to determine the exact number of attacking aircraft in a radar track presentation. Passive search , tracking and engagement is also possible with the NTAS. The NTAS makes it possible to fire AMRAAM's totally silently without radar emission. No performance data is available from open sources about the NTAS.

IRST systems are clearly an area of intense research, but performance data are scant in open sources (Cullen, 1998). We have selected some typical systems as a baseline for assumed performance. They are either in development or in the early phases of operational testing or use. Table 2 summarizes some available performance data.

36

The source does not indicate exactly what is meant by maximum range. One can only speculate that a target with a signature commensurate with the signature of a subsonic sea-skimming missile will be detected at this range with some probability under given (most likely ideal) atmospheric conditions.

	SAGEM VAMPIR (DIBV-1A)	Lockheed Martin AADEOS Advanced Air Defense Electro-Optical Sensor	Hollandse Signaalapparaten Spar Aerospace SIRIUS
Range Fighter (km)	10-18	NA	30
Range Supersonic Missile (km)	14-27	NA	35 (sea-skimmer)
Range Subsonic Missile (km)	9-16	NA	21 (sea-skimmer)
Number of targets tracked	50	256 in track while scan mode	NA
Spectral Band(s) (μm)	3-5 and 8-12	3-5 and 8-12	3-5 and 8-12
Operational Status	In operational use	Demo model built and delivered to USA in 1991	Pre-production model scheduled for 1998/99

Table. 2 IRST Performance

As an example, for the Pilkington Optronics ADAD air defense sensor, Jane's states that "The manufacturer claims that it increases the chances of a successful kill by 400 per cent" (O'Leary, 1997). The sensors are all made to be integrated into systems using active sensors. For an example the SIRIUS, "can be integrated with any combat system, in close cooperation with any sensor system, ranging from simple track radar to autonomous close in weapons systems up to active phased-array radar." Clearly, the military planner cannot take claims such as the one mentioned above on face value. We now go on to build two simulation models to get an understanding for the required performance parameters for IRST sensors.

B. SCENARIO AND SIMULATION

NASAMS is typically deployed in the defense of a single airbase. Oerland Main Air Base located in central Norway was chosen as the basis of a notional scenario. This airbase is located at the Atlantic coastline, west of the city of Trondheim. Rugged terrain, fjords, and a very large number of small and large islands dominate the area. A Cartesian coordinate system was superimposed on a line-drawing of the area. The notional scenario includes the different NASAMS elements deployed to defend an airbase located at (115,120,0). Figure 8 shows a drawing of the scenario and deployment of NASAMS units. Some of the units somewhat offset from their true positions for clarity.

Figure 8. NASAMS Deployment

The deployment is roughly based on known practices. It is assumed that the airbase will be attacked by cruise missiles (CM) with low radar cross section. All cruise missiles approach from the west, but fly trough randomly chosen initial points before reaching the target. The measures of effectiveness are approximated by calculations on data collected from a random sample of attacking cruise missiles.

The IRST and LASR components are of the "cookie cutter" type. When a target is inside the detection range the target is considered detected. Furthermore, all sensors and other units in system are "on the network" and are assumed to exchange data instantaneously. The influence of earth curvature ignored. NASAMS would usually be connected into the NATO Air Defense Ground Environment (NADGE). In the model it is assumed that the larger air defense network is unable to provide any early warning for the targets of interest. Also, NASAMS would be connected to several short-range air defense weapons. In the model only the effect of the AMRAAM launchers is considered.

Three measures of performance are used to quantify the effect of an IRST network: mean warning time, minimum cumulative time in action volume, and the number of successful missile flights. One of the most crucial elements in air defense success is warning-time of an impending attack. The more warning-time the better. An IADS at high levels of readiness consumes resources at a very prolific rate. High levels of readiness cannot be maintained for very long periods of time. Warning time allows the IADS to manage its resources by changing its states of readiness. The precise relation between warning time and system effectiveness will vary from system to system, but, in general it is assumed that system performance improves with increasing warning time. One of the great benefits of the NASAM system is that engagement is not dependent on target illumination for missile guidance. It suffices for the target to be tracked by one of the networked sensors until the AMRAAM missile can turn on its internal radar. We can define "Action Volume" as that part of the volume in which a target can both be tracked by some sensor and is within range of a weapon system, in our case the AMRAAM launchers. A target can be engaged only if it is simultaneously tracked and within range of some launcher. The cumulative time that a specific target profile is within the action volume, as measured from the target impact point, will be an indicator of the ability to successfully engage that target. In the second model we measure the mean number of successful missile flights. In this model the AMRAAM

average speed is set to 410 m/s. A speed of 250 m/s is used for the cruise missile for both models.

C.   FIRST MODEL

Figure 9 illustrates how the components have been connected for the simulation model. As before the lightning bolts in the figure indicate both property and event connections between components. For clarity only one of the connections have been shown between the components, for example, the tracking correlator is connected to each LASR, but only one connection is shown.
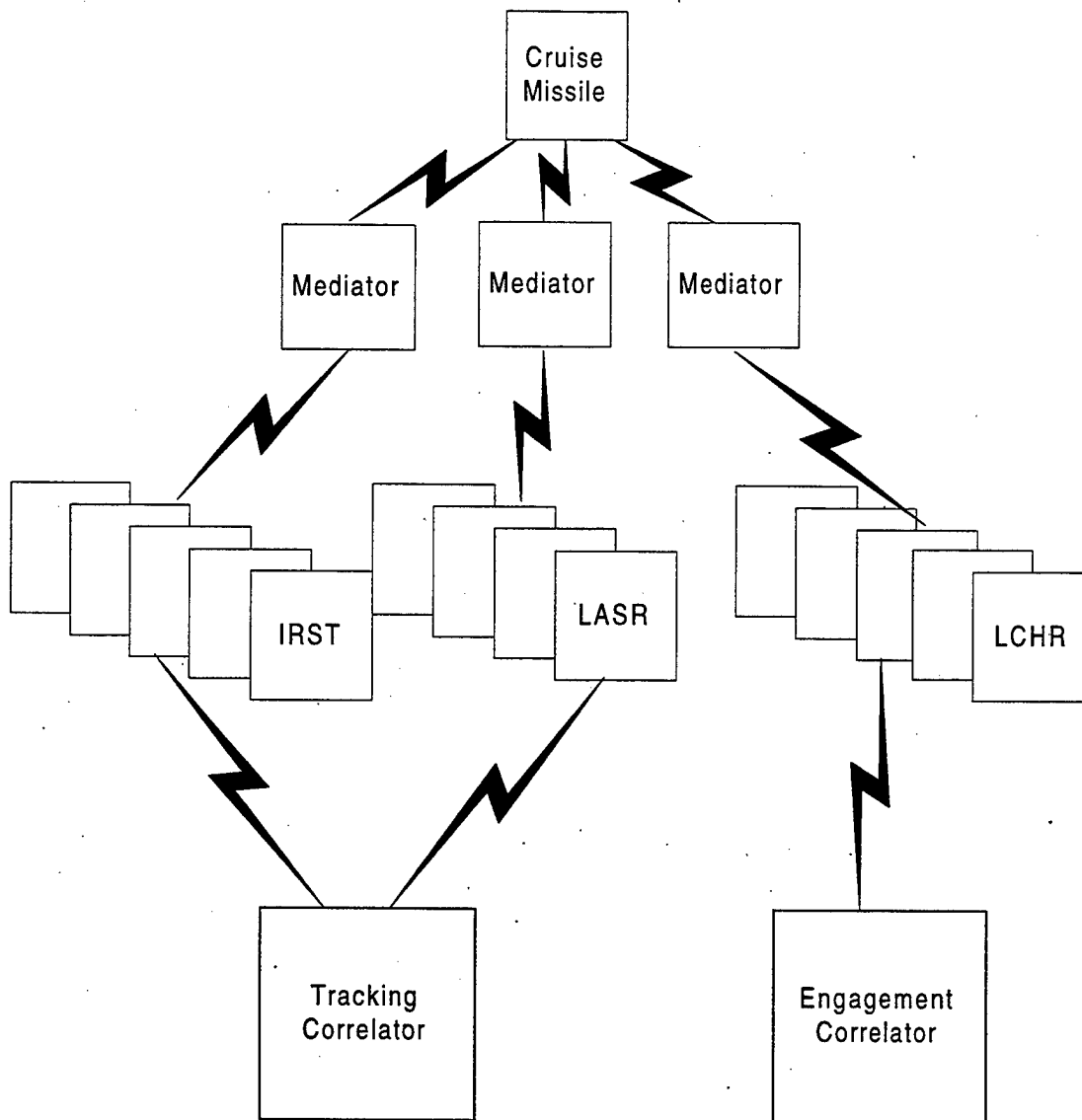
Figure 9. Components and Connection for First Model

The cruise missile is set to move towards the target. On its way it may encounter the different NASAMS elements and the IRST's. The role of the mediator components is to ensure correct interaction with the cruise missile and each of the NASAMS elements. The tracking correlator component "listens" to each of the sensors (both LASR and IRST) and determines when the cruise missile is tracked by at least one sensor. The tracking correlator is itself an event source for tracking events. The functioning of the engagement correlator is similar, but it is connected to the launchers

to keep track when the cruise missile is inside of one or more of the AMRAAM launch envelopes. The simulation consists of gathering data on when the cruise missile is first detected, when it is tracked, and when it is both tracked and within range of one or more of the AMRAAM launchers. In Figure 10 average warning time is plotted against IRST and LASR detection ranges.
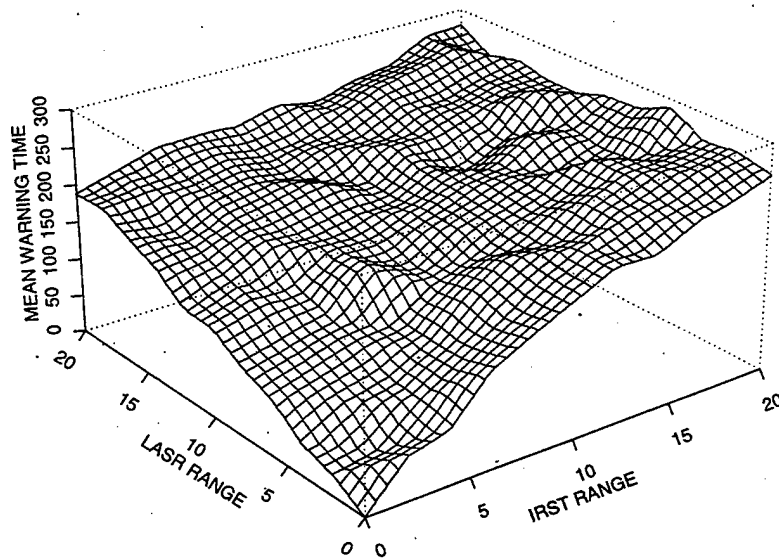

Figure 10. Mean Warning Time versus LASR/IRST Range

Cullen (1989) gives a detection range of about 40 km for NASAMS versus a so-called low radar cross section fighter-sized aircraft. It is reasonable to expect this range to fall by an order of magnitude versus a stealthy (low RCS) cruise missile (Knight, 1989, pp. 154). According to the data on IRST systems typical detection ranges for a low flying subsonic cruise missile would range from 10 to 20 km. If we assume that the LASR detection range is in the area between 0 and 10 km it is apparent from the surface plot that the network of IRST has a very positive effect on average warning time. On the

other hand, when the LASR detection range approaches the 15-20 km range the effect of the IRST sensors diminishes. Figure 11 displays average warning time versus range for the fixed LASR detection range of 6 km.
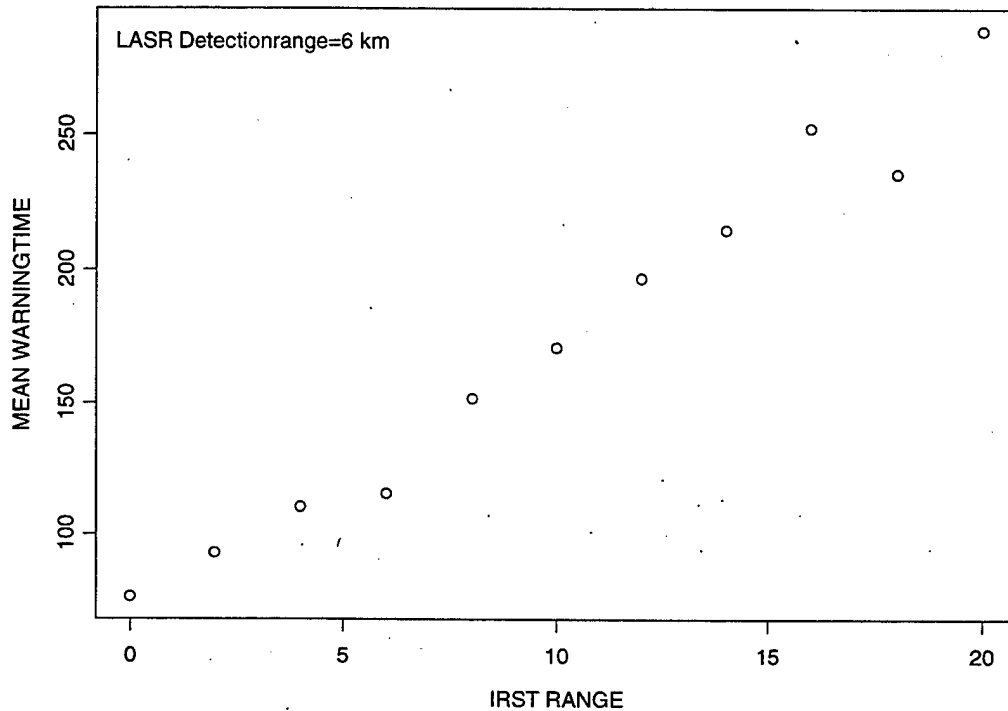

Figure 11. Average Warning Time as Function of IRST Range

The average warning time goes from 75 seconds to 290 seconds as the IRST range goes from 0 to 20 km. The relationship seems to be fairly linear. Exactly what impact this increase would have is subject to system specific data on readiness states and transitions that are not accessible. However, we can conclude that IRST's seems to have a very noticeable effect on average warning times for relevant LASR ranges, and for IRST ranges within those claimed by the IRST manufacturers.

The reader may wonder why there is a sudden "loss" in range at IRST range 18 since the curve otherwise seems quite linear. It must be borne in mind that each

combination of LASR and IRST range is "covered" by a small sample of 100 cruise missiles, and so the mean warning time is a random variable. As a selected example we looked at the data for LASR range=6 and IRST range=18, and performed a bootstrap with respect to the mean of the 100 values gathered from the simulation (Efron,1993). The estimated distribution of the mean warning time for this range combination is shown in Figure 12.
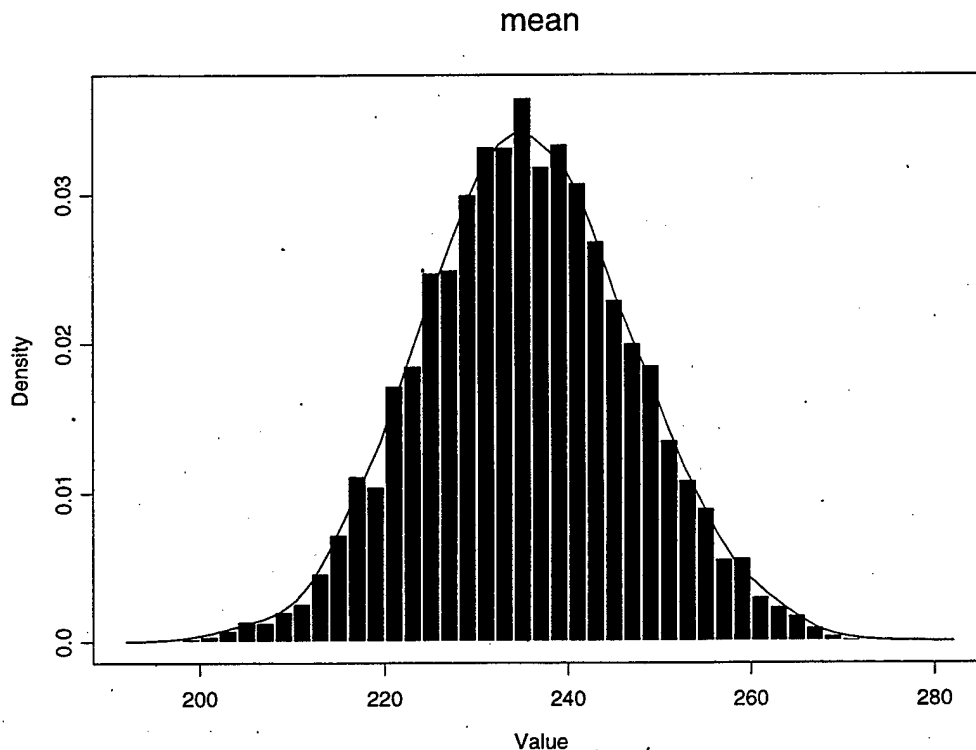
mean

Figure 12. Bootstrap Estimate of Distribution of Mean Warning Time

The observed value for the mean is 235.5, and the bootstrap yielded an estimated 95% BCA confidence interval for the mean warning time for this range combination to be from 215.6 to 260.9. Using the Central Limit Theorem we find an estimated 95% CI to be from 212.3 to 258.7. Thus this sudden "fall" in range is well within what we must have to expect from chance alone given the relatively small sample size. Figure 13 displays the second MOE, minimum cumulative time in action volume.
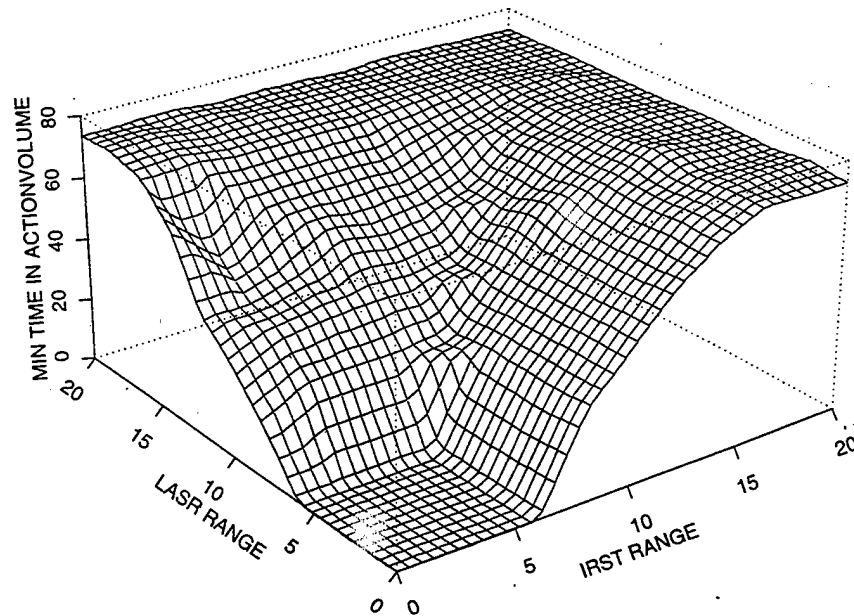
45

Figure 13. Minimu ...umulati ...me In Action Volume

NASAMS can engage a target if it is tracked (by at least one of the sensors in the system) and inside one of the launcher ranges. Minimum cumulative time inside the action volume is an indicator for the ability to engage the target. As we can see from the plot the response is zero for all combinations of LASR and IRST ranges from 0 to 6 km, when we suddenly start getting a response. What this means is that of the sample of 100 missiles for each range combination, at least one had a zero cumulative time in the action volume.

D.   EXTENDED MODEL

The second model is an extension of the first model that adds a component to simulate surface-to-air missile flights. When a cruise missile is tracked and enters a launcher's max range a missile is flown out to intercept the cruise missile. The parameters for the simulated AMRAAM are an average speed of 410 m/s and a max flight time of 60 seconds. Experimentation showed that this combination of parameters gives an approximate max range of 15 km versus a target with a speed of 250 m/s when the missile is fired as the target has the launcher exactly on the beam. Also, this estimated average speed is consistent with the missiles given max ballistic range of 22 km under the assumption of a 60 seconds max flight time.

The only data collected in this model are the number of successful missile flights per cruise missile attacking. Figure 14 shows the components and connections for the second model. A component has been added to act as a fire distributor. This component listens to the tracking and engagement correlators. If a cruise missile is tracked when it enters a launch zone the fire distributor fires a simulated AMRAAM component at the target. No missile is fired if tracking occurs after the cruise missile has entered a launch zone.
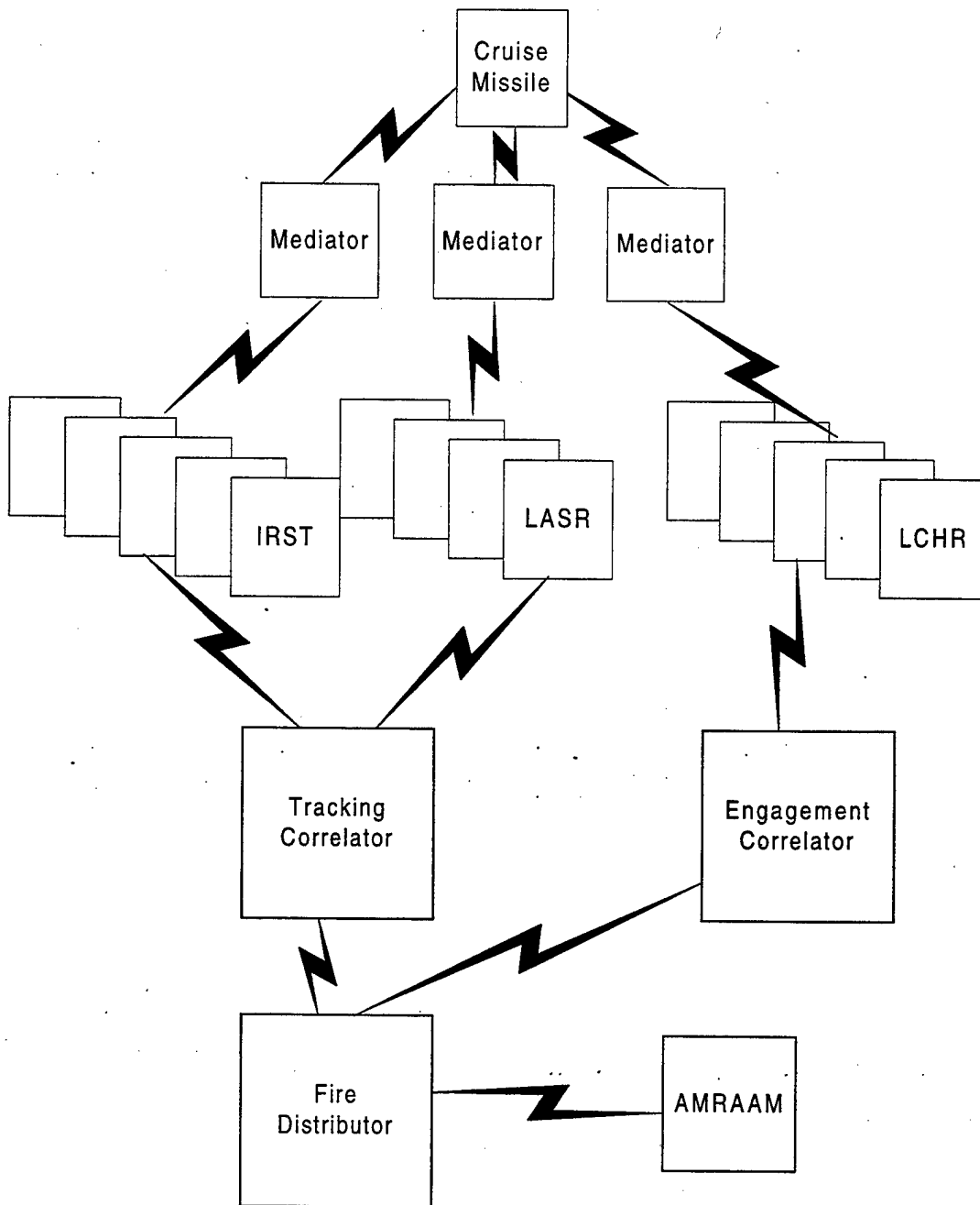
Figure 14. Extended Model

Figure 15 shows a plot of mean number of successful missile flights versus IRST and LASR ranges.
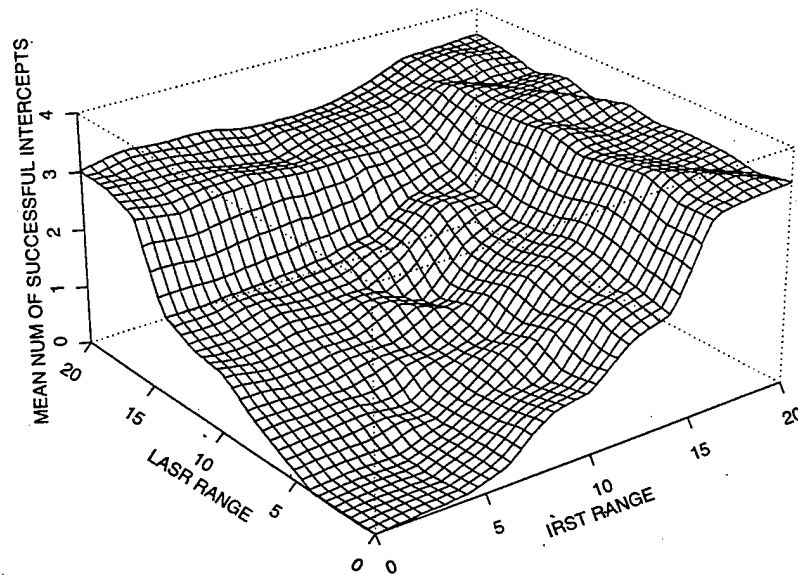

Figure 15. Mean Number of Successful Missile Flights

The response is almost zero until the range reaches around 6 km. From this point on we get an increasing number of successful missile flights. Notice that we would have drawn a similar conclusion if we had used the minimum cumulative time in action volume as our MOE. Figure 16 shows the minimum number of successful intercepts versus the ranges.
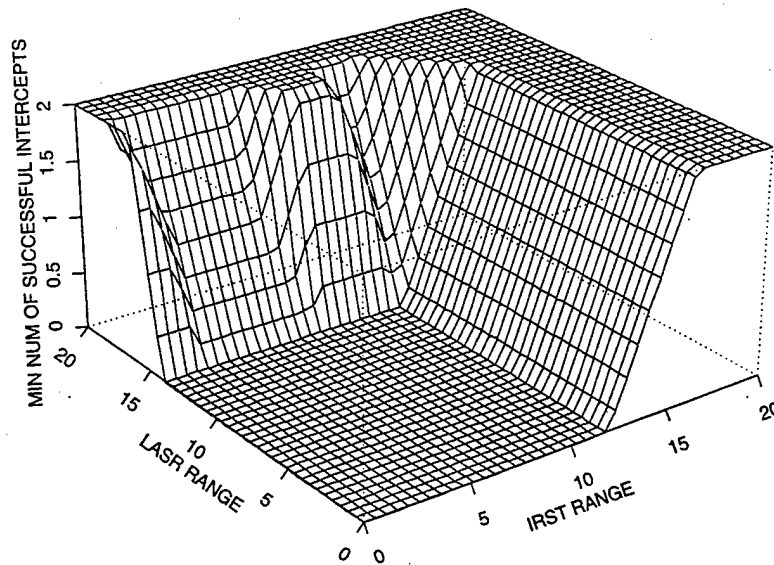
Figure 16. Minimum Number of Successful Missile Flights

It is interesting to compare Figure 16 to Figure 13, the plot of minimum cumulative time in action volume. As we can see from the plot, the minimum number of successful intercepts jumps from 0 to 2 at around 12 km range for the IRST and LASR. What this means for the ranges up to around 12 km is that there was at least one out of 100 missiles for which there was no successful missile flight. From Figure 13 we can see that an IRST range of 10 to 15 km corresponds to a cumulative time in action volume from 20 to 40 seconds. According to open sources NASAM needs a minimum of about 10 seconds from tracking commences until the first missile can be fired. To this we must add the time it takes to establish a firm track on the target. From Figure 13 we can observe that an IRST range of 12 km corresponds to approximately 20 seconds minimum cumulative time in action volume.

E.   SIMULATION RESULTS

A typical deployment of a NASAMS battery in the defense of an airbase located on the Norwegian coastline was chosen as the basis for a simulation model to study the effect of a network of IRST sensors. Since the study aims to get a rough order of magnitude estimate of the potential impact of IRST, a typical deployment was assumed to be representative. Target acquisition and tracking is fundamental to all air defense systems. The first model therefore concentrates almost exclusively on these aspects of the problem.

Performance parameters for IRST sensors and reduction of radar detection ranges due to stealth was estimated using open sources. A network of 12 conceptual IRST sensors was deployed. The exact number of sensors that can be used will be subject to factors such as procurement and operating costs. Twelve was chosen as reasonable number since at this level the sensors can be co-located with the different NASAM components, requiring no new and expensive infrastructure. The maximum detection-ranges given in Table 2 are likely to be too optimistic given that the simulation is based on detection with probability 1 at max range.

Warning time was defined to be the time from first detection of an attacking missile to its impact on the airbase. The first simulation model shows that the chosen IRST network has a clear positive effect on average warning time. For example, a typical detection range for a stealthy cruise missile could be in the range of 6 km. The average warning time in the radar-only case is 75 seconds. With the addition of the IRST network with an average detection range of 10 km this time was doubled. At an IRST range of 15 km the average warning time is in the range of 250 seconds. Although we lack detailed knowledge of the required transition times between readiness levels for NASAMS, it is quite clear from the numbers involved that the effect of the IRST network is significant. The major factor in this result is the choice of the airbase as the only target. Since the bulk of the IRST sensors are located on the perimeter of the

airbase, even small IRST ranges will yield positive effects on the average warning time. Both the Vampire and Sirius sensor systems (Table 2) have the required performance level to yield good results with regard to this MOE.

The second MOE, minimum time in action volume was also estimated with the first model. This MOE indicates how much opportunity the NASAM system has to engage an incoming target. Figure 12 shows that the response is zero up to an IRST range of 6 km. This means that out of a random sample of 100 attack profiles, at least one had a minimum cumulative time in action volume of zero, leaving NASAM without any engagement opportunity. However, it was estimated that a minimum time of about 30 seconds should correspond to an engagement opportunity. In that case the IRST range must be at least 10 to 15 km to compensate for the low radar detection range. This takes us to about the maximum range for the Vampir, but the Sirius with its claimed 21 km maximum range would yield very good results.

In the extended model, components were added to simulate actual AMRAAM firings and intercepts. If an attacking cruise missile was tracked when it entered a launchers max range the intercept was simulated. Data were collected on the number of successful intercepts. For LASR detection ranges of between 0 and 12 km there is a jump from 0 to 2 at around 12 km IRST range. The conclusions drawn from this are the same as those for the second MOE, minimum cumulative time in the action volume.

If we consider the simulation models to realistically capture the most important factors determining NASAMS engagement capabilities in the given scenario, we must conclude that the required IRST performance is at the outer edge of what the selected IRST systems can provide. The oldest of the systems, the Vampir, with a detection-range of 9-16 km is at the lower end of the desired range. On the other hand, the Sirius, which promises to be at the pre-production stage in 1998/99, has the required performance, perhaps even with a slight margin. Therefore, adding an IRST network to

NASAMS could become an operationally effective option to compensate for the lack of performance versus low RCS targets in the near future.

F.   SUGGESTIONS FOR FURTHER SIMULATION WORK

Precision guided munitions such as HARM has been around for quite some time. Therefore, it is perhaps doubtful that the airbase will be the only target. Threatened by such weapons, the LASR's may be forced to shut down for shorter or longer periods to change positions. In that case an IRST network would, of course, be of great interest. It would benefit the analysis to simulate an operational pattern of movement for the LASR's. We would then be able to deduce what effect the IRST systems can have to compensate for the total absence of one or more LASR's at any one time. If this study concluded positively for the IRST systems there would be a strong case to look at their inclusion into the NASAM system. This addition to the simulation model should be as easy to carry out as the move from the first to the extended model. All that is required is to make a composite component consisting of an existing LASR component and an existing RouteMover.

V. DISCUSSION

A brief look at history showed us that technology has had a major impact on the efficiency of air defense operations. Since air defense is a purely reactive form of warfare, the application of scientific principles to the design and deployment of air defense systems is a major factor in achieving effectiveness. Today's air defense planners face rapidly changing technological developments, both for offensive weapons and for sensors. Understanding the impact of technology on air defense operations must be done continually and at an increasing pace. The combination of dwindling defense resources and rapid technological developments makes the need for analysis more critical. Yet with current software architectures, even the analysis activity may be prohibitively costly for small nations. The planner needs access to tools that can be used for basic exploration, analysis and planning. Simulation is the most used tool in that regard. Building simulation models has traditionally been an expensive and difficult process, quite out of reach for the individual planner or small workgroup. In most cases simulation models must be programmed from scratch. A new object-oriented programming language called Java was introduced in the mid 1990's. Java is perhaps the first programming language that allows even amateur programmers to produce truly useful and maintainable software. Although Java has a software component standard called Java Beans, for the most part it is used for building Graphical User Interfaces, a job that it does very nicely. However, the requirements imposed by simulation are more general, and in this thesis a software component standard was developed specifically as a baseline for simulation components. A component library was constructed that was small but very useful.

A case study was carried out on the impact of IRST sensors on air defense using simulation and the component library. We started with three simple components and showed a demonstration of their basic use. Even though the components were simple, they turned out to be good enough for practical analysis in a realistic scenario. The

components for simulating radar, IRST sensors and AMRAAM launchers were all easily derived from a more basic component. Simulation components for a cruise missile and an AMRAAM were similarly derived from a basic component for movement. In this we benefited fully from Java's object orientation, reusing the implementation of more basic components.

The model used had one cruise missile, four LASR's, nine LCHR's and 12 IRST sensors. The cruise missile interacted with each of those entities for a total of 25 interactions. Since separate mediators handle interactions, scaling the model up from 1 to 25 interactions (or any other number) was simply a question of connecting components. Adding new components and interactions to an existing model did not require modification of the existing model, a feature not available in any other software architecture known to the author. Even if Java is the technology enabling components, the scalability is an effect of the software component approach more than any features of Java. The author used no more time and effort in building and running the models demonstrated here than what would be available in a real life, non-academic situation. Developing the Modkit software component architecture took the better part of a year. However, this was a one-time effort, since Modkit has a high degree of reusability.

Progress in almost all fields of human endeavor, including simulation modeling, can best be made in small steps. The modeling effort that started with a very simple model could be run immediately and, as a result, initial insights could be used to make further model design decisions, a very desirable feature in a situation characterized by complexity and uncertainty. Existing software engineering schemes can handle development of software for stable and known situations well, but require that models be nearly finished before they can be exercised, precluding the feature of stepwise model development offered by the component architecture.

Collecting data from the model was very simple thanks to the use of events. We only needed to add components that can listen to the events fired by the different

components in the model. In fact, the components in the model are completely ignorant that data is being gathered. This saves much effort, since not one single line of code in the simulation components must be rewritten to collect disparate or unanticipated data from the model. The extended model was built from the first model by adding two components. This simple addition gave us a much more elaborate model in which we could study the actual engagement opportunities by carrying out intercepts. Whereas the extended model was more computationally expensive, it did not yield much new information. It may well be that the planner would chose to work with the first model. If so, very little effort would have been expended deducing this fact. This contrasts very favorably to starting with a complex model and shaving off parts that most likely will not be useful. In short, it is possible to reduce wasted effort by reducing the size of the initial model, since we could use the model for analytic purposes immediately. This is especially critical in a situation where it is difficult to decide before the fact what the sensible MOE's should be.

High tempo seems to be a dominating feature of theories of modern warfare. If simulation models are to be used for planning purposes under such circumstances the cycle time from one model to the next must be very short. Current methods fall far short of the requirements. We have shown how the flexibility and scalability of the Modkit component architecture reduces modeling cycle-time dramatically in the context of air defense planning. Modkit can best be thought of as an extension of the Java programming language . Combined with the even higher level abstractions and reusability achieved with the Modkit component architecture this allows the functions of domain expert and simulation analyst to be combined in one individual.

VI. CONCLUSIONS

This research started as an effort to quantify the impact of passive sensors on the effectiveness of integrated air defense systems. System simulation was found to be the only available analysis technique. Realizing that the task was too large for the time and resources available, a software component system was developed to provide flexibility and growth potential to the required simulation model.

Literature research has shown that the main features of our components have much in common with the feature of the Structured Modeling framework. We arrived at our component system via an empirical method.

A practical scenario comprising a modern Medium Range Surface to Air System (MSAM) was laid out as the basis for the simulation model. The impact of a network of IRST sensors on warning time and engagement opportunities in the MSAM was approximated for a set of IRST and active radar detection ranges. The gathered data indicate that IRST systems could be valuable in the near future.

Military planners and analysts should be able to build a very broad range of simulation models using a library of reusable components. This requires that a deeper understanding of software component technology and standards be developed. We have discussed how component software technology can have a major impact on the efficiency of simulation modeling and pointed to research and literature showing the great interest in this topic, both civilian and military.

Working with software components is different from traditional software engineering and design, and the methods of documentation are different. We have given examples on how software components may be documented, and we have pointed to how this method of documentation also can serve as a suitable means of communication between component users and component programmers.

59

One of the main features of a simulation is to imitate the behavior of entities and their interaction. Interactions are a source of complexity. We have shown how to tackle interaction complexity by a divide-and-conquer method called the mediator. In addition, this method provides stability, reusability of components, and scalability of models.

In this thesis the feasibility of building a simulation model based on a loosely coupled (software) component approach has been demonstrated in practice. The experts agree that this approach has great promise, but also requires high component quality. This applies to both the abstract aspects of the components and their implementation. The required level of quality of components or implementation has certainly not been reached in this thesis. Yet the small-scale success of the component approach demonstrated here, together with the widespread and general success of component technology indicates that great benefits can be reaped from applying the component approach to real-life simulation models for air defense analysis.

LIST OF REFERENCES

Blanning, R. W., "Special Issue on Model Management Systems," *Decision Support Systems* 9, 1993.

Bradley, G.H., Buss A.H., *An Architecture for Dynamic Planning Systems Using Loosely Coupled Components,* Proposal For Reimbursable Research, Naval Postgraduate School Monterey, CA, 1997.

Brown, A.W., *Component-Based Software Engineering*, Selected Papers From The Software Engineering Institute, IEEE Computer Society Press, Los Alamitos, CA, 1996.

Cooling, F. B., *Case Studies in the Achievement of Air Superiority*, U.S. Government Printing Office, Washington, D.C., 1994

Cullen, T., *Jane's Land-Based Air Defense*, Eleventh Edition, 1998-99.

Creveld, M., *Technology and War, From 2000 BC to the Present*, New York, Free Press, 1989.

Davis, P., *An Introduction to Variable-Resolution Modeling and Cross-Resolution Model Connection*, RAND, Santa Monica, CA, 1993.

(DMSO) Defense Modeling and Simulation Office, Defense and Simulation Initiative, Office of the Director, Defense and Engineering, Department of Defense, 1995.

Efron, B., *An Introduction to the Bootstrap*, Chapman and Hall, 1993.

Geoffrion, A., "Structured Modeling," *Encyclopedia of Operations Research and Management Science*, pp. 652-655, Kluwer, Academic Press 1996.

Friedman, G., *The Future of War, Power, Technology and American World Dominance in the 21'st Century*, Crown Publishers, New York, 1996.

Gamma, E., et al., *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley 1995.

Hallion, R., *Storm over Iraq, Air Power and the Gulf War*, Smithsonian Institution Press, Washington and London, 1991.

Jacobson, I.,*Object-Oriented Software Engineering, a Use Case Driven Approach*, Addison-Wesley, 1993.

Knight, M., *Strategic Offensive Air Operations*, Brassey's UK, 1989.

Krishnan, R., "Model Management", *Encyclopedia of Operations Research and Management Science*, pp. 400-403, Kluwer, Academic Press 1996.

O'Leary, A.P., *Jane's Electro Optic System*, Third Edition, 1997-98.

Szyperski, C.,*Component Software, Beyond Object-Oriented Programming*, ACM Press New York, 1997.

National Research Council (U.S.). Naval Studies Board. Committee on Technology for Future Naval Forces, *Technology for the United States Navy and Marine Corps, 2000-2035 : becoming a 21st-century force*, Washington, D.C., National Academy Press, 1997.

Terraine, J., *The Right of the Line, the Royal Air Force in the European War 1939-1945*, Hodder and Stoughton, 1988.

Zeigler, B. P., *Theory of Modeling and Simulation*, Wiley, New York, 1976.

APPENDIX A.   JAVA CODE FOR ATOMIC DEMO

```java
/**
* @author Arent Arntzen
* Test of Atomic interaction
* Started 5 Sep 98
*/

import modkit.*;
import modutil.spatial.*;
import modsim.*;
import simkit.*;

public class AtomicTest {

    public static void main(String[] args) {
        Coor3D[] xwpts=new Coor3D[] { new Coor3D(10.0,0,0),
                                      new Coor3D(0,0,0),
                                      new Coor3D(-10.0,0.0,0.0)};
        Route4D xRoute;
        xRoute=new Route4D(xwpts,1.0);
        RouteMover xMover=new RouteMover("xMover");
        xMover.setMaxSpeed(2.0);
        xMover.setRoute(xRoute);
        xMover.go();
        ModEventListener tbf=new TabbedFrameModEventListener();
        xMover.addModEventListener(tbf);
        BasicSensor xSensor=new BasicSensor("xSensor",true);
        xSensor.addModEventListener(tbf);
        xSensor.setMaxRange(1);
        SensorMoverMediator SMM=new
        SensorMoverMediator("SMM",xSensor,xMover);
        SMM.setMinTimeStep(0.01);
        Schedule.startSimulation();
    }
}
```

APPENDIX B. JAVA CODE FOR COMPOSITE DEMO

```java
/**
 * @author Arent Arntzen
 * 5 Sep 98
 * Test of composite interaction.
 */

import modkit.*;
import modsim.*;
import modutil.spatial.*;
import simkit.*;

public class CompositeTest {

    public static void main(String[] args) {
        Coor3D[] xwpts=new Coor3D[] { new Coor3D(10.0,0,0),
                                      new Coor3D(0,0,0),
                                      new Coor3D(-10.0,0.0,0.0)};
        Coor3D[] ywpts=new Coor3D[] { new Coor3D(0,10.0,0),
                                      new Coor3D(0,0,0),
                                      new Coor3D(0,-10.0,0)};
        Route4D xRoute=new Route4D(xwpts,1.0);
        Route4D yRoute=new Route4D(ywpts,1.0);;
        MovingSensor ms1=new MovingSensor("X");
        ms1.init(xRoute,2.0,1.0);
        MovingSensor ms2=new MovingSensor("Y");
        ms2.init(yRoute,2.0,2.0);
        ModEventListener tbf=new TabbedFrameModEventListener();
        ms1.addModEventListener(tbf);
        ms2.addModEventListener(tbf);
        SensorMoverMediator xy=new
        SensorMoverMediator("XYMediator",ms1,ms2);
        SensorMoverMediator yx=new
        SensorMoverMediator("YXMediator",ms2,ms1);
        xy.addModEventListener(tbf);
        yx.addModEventListener(tbf);
        yx.setMinTimeStep(0.01);
        xy.setMinTimeStep(0.01);
        Schedule.startSimulation();
    }
}
```

APPENDIX C. JAVA CODE FOR FIRST SIMULATION MODEL

```java
/**
* @author Arent Arntzen
* Started 1 Aug 98
*/

package models.iads;

import modkit.*;
import modsim.*;
import modutil.spatial.*;
import simkit.*;
import java.text.DecimalFormat;

public class FirstModel {
    private static DecimalFormat df = new DecimalFormat("#.0");
    public static void flyMissile(int    numMissiles,
                                  double IRSTrange,
                                  double LASRrange,
                                  double LCHRrange) {
/**
* Make a factfinder to collect statistics
*/
      FactFinder ff=new FactFinder();
/**
* Make some sensors of the IRST type
*/
      IRST[] irsts=IADSutils.makeIRSTarray(
      new String[] {"IRST-1","IRST-2","IRST-3","IRST-4","IRST-5",
                    "IRST-6","IRST-7","IRST-8","IRST-9","IRST-10",
                    "IRST-11","IRST-12"},
      IRSTrange,
      new Coor3D[] {new Coor3D(130,130,0),new Coor3D(110,125,0),
                    new Coor3D(110,115,0),new Coor3D(85,125,0),
                    new Coor3D(90,145,0),new Coor3D(85,100,0),
                    new Coor3D(140,130,0),new Coor3D(140,150,0),
                    new Coor3D(150,160,0),new Coor3D(20,120,0),
                    new Coor3D(60,120,0),new Coor3D(20,155,0)});
/**
* Make the four LASR's
*/
      StaticCC[] lasrs=IADSutils.makeLASRarray(
      new String[] {"LASR-1-Orland","LASR-2-Valset","LASR-3-
Nord","LASR-4-Tarva"},
      LASRrange,
      new Coor3D[] {new Coor3D(110,125,0),new Coor3D(75,100,0),
                    new Coor3D(145,155,0),new Coor3D(85,145,0)});

/**
* Make the LHCR's...
*/
      LCHR[] lchrs=IADSutils.makeLCHRarray(
      new String[] {"LCHR-1-OrlandOst","LCHR-2-OrlandNord",
                    "LCHR-3-OrlandSyd","LCHR-4-Storfosna",
```

67

```
                        "LCHR-5-Tarva","LCHR-6-Valset",
                        "LCHR-7-Nord1","LCHR-8-Nord2",
                        "LCHR-9-Nord3",},
        LCHRrange,
        new Coor3D[] {new Coor3D(130,130,0),new Coor3D(110,125,0),
                        new Coor3D(110,115,0), new Coor3D(85,125,0),
                        new Coor3D(90,145,0),   new Coor3D(85,100,0),
                        new Coor3D(140,135,0),new Coor3D(140,150,0),
                        new Coor3D(150,160,0)});
/**
* Factfinder listen to all
*/
        for(int i=0;i<irsts.length;i++) {
            irsts[i].addModEventListener(ff);
        }

        for(int i=0;i<lasrs.length;i++) {
            lasrs[i].addModEventListener(ff);
        }

        for(int i=0;i<lchrs.length;i++) {
            lchrs[i].addModEventListener(ff);
        }

        RouteMover cm=new RouteMover("CruiseMissile");
        cm.setMaxSpeed(0.25);
/**
* Listen to the cruisemissile also
*/
        cm.addModEventListener(ff);

/**
* Make and connect the mediators for the LASR's
*/
        for(int i=0;i<lasrs.length;i++) {
            SensorMoverMediator cmVsLasr;
            cmVsLasr=new SensorMoverMediator("lasr-Vs-CM",lasrs[i],cm);
            cmVsLasr.setMinTimeStep(0.01);
        }

/**
* Make and connect the mediators for the LCHR's
*/
        for(int i=0;i<lchrs.length;i++) {
            SensorMoverMediator cmVsLchr;
            cmVsLchr=new LauncherMoverMediator("lchr-Vs-CM",lchrs[i],cm);
            cmVsLchr.setMinTimeStep(0.01);
        }

/**
* Make and connect the mediators for the IRST's
*/
        for(int i=0;i<irsts.length;i++) {
            SensorMoverMediator cmVsIrst;
            cmVsIrst=new SensorMoverMediator("IRST-Vs-CM",irsts[i],cm);
            cmVsIrst.setMinTimeStep(0.01);
        }
```

```java
        for(int i=0;i<numMissiles;i++) {
            ff.reset();
            Schedule.reset();
            Route4D r=IADSutils.randomRoute2( new Coor3D[] {new
                        Coor3D(115,120,0)});
            cm.setRoute(r);
            cm.go();
            Schedule.startSimulation();
            System.out.println( df.format(IRSTrange)+"\t"+
                                df.format(LASRrange)+"\t"+

        df.format(LCHRrange)+"\t"+ff.toDataOnly());


        }
    }

    public static void main(String[] args) {

        System.out.println("IRST"+"\t"+"LASR"+"\t"+"LCHR"+"\t"+"WT"+"\t"+
        "TT"+"\t"+
                "LCT"+"\t"+"AVT"+"\n");
        for(int ir=0;ir <= 20;ir+=2) {
            for(int la=0; la<=20; la+=2) {
                flyMissile(100,ir,la,15.0);
            }
        }
    }
}
```

APPENDIX D. JAVA CODE FOR EXTENDED MODEL

```java
/**
* @author Arent Arntzen
* Started 1 Aug 98
*/
package models.iads;
import modkit.*;
import modsim.*;
import modutil.spatial.*;
import simkit.*;
import simkit.data.*;
import java.text.DecimalFormat;

public class ExtendedModel {
    private static DecimalFormat df = new DecimalFormat("#.0");
    public static void simulate(int numMissiles,
                                double IRSTrange,
                                double LASRrange,
                                double LCHRrange,
                                double minTstep) {
      TrackCorrelator tc=new TrackCorrelator("TrackCorrelator");
      FireDistributor fd=new FireDistributor("FDC",tc,0.5,0.1,60.0);
      FuzeFunctionCounter ffc=new FuzeFunctionCounter("ffc");
      SimStopper sstop=new SimStopper("Stopper");
      RouteMover cm=new RouteMover("CruiseMissile");
      cm.setMaxSpeed(0.25);
      cm.addModEventListener(sstop);
      fd.addModEventListener(ffc);
      StaticCC[] lasrs=IADSutils.makeLASRarray(
      new String[] {"LASR-1-Orland","LASR-2-Valset","LASR-3-
Nord","LASR-4-Tarva"},
      LASRrange,
      new Coor3D[] {new Coor3D(110,125,0),new Coor3D(75,100,0),
                    new Coor3D(145,155,0),new Coor3D(85,145,0)});
      LCHR[] lchrs=IADSutils.makeLCHRarray(
      new String[] {"LCHR-1-OrlandOst","LCHR-2-OrlandNord",
                    "LCHR-3-OrlandSyd","LCHR-4-Storfosna",
                    "LCHR-5-Tarva","LCHR-6-Valset",
                    "LCHR-7-Nord1","LCHR-8-Nord2",
                    "LCHR-9-Nord3",},
      LCHRrange,
      new Coor3D[] {new Coor3D(130,130,0),new Coor3D(110,125,0),
                    new Coor3D(110,115,0), new Coor3D(85,125,0),
                    new Coor3D(90,145,0),  new Coor3D(85,100,0),
                    new Coor3D(140,135,0),new Coor3D(140,150,0),
                    new Coor3D(150,160,0)});
      IRST[] irsts=IADSutils.makeIRSTarray(

      new String[] {"IRST-1","IRST-2","IRST-3","IRST-4","IRST-5",
                    "IRST-6","IRST-7","IRST-8","IRST-9","IRST-10",
                    "IRST-11","IRST-12"},
      IRSTrange,
      new Coor3D[] {new Coor3D(130,130,0),new Coor3D(110,125,0),
                    new Coor3D(110,115,0),new Coor3D(85,125,0),
```

```
                              new Coor3D(90,145,0),new Coor3D(85,100,0),
                              new Coor3D(140,130,0),new Coor3D(140,150,0),
                              new Coor3D(150,160,0),new Coor3D(20,120,0),
                              new Coor3D(60,120,0),new Coor3D(20,155,0)});

    /**
    * Make and connect the mediators for the LASR's
    */
            for(int i=0;i<lasrs.length;i++) {
                SensorMoverMediator cmVsLasr;
                cmVsLasr=new SensorMoverMediator("lasr-Vs-CM",lasrs[i],cm);
                cmVsLasr.setMinTimeStep(minTstep);
            }
    /**
    * Make and connect the mediators for the IRST's
    */      for(int i=0;i<irsts.length;i++) {
                SensorMoverMediator cmVsLasr;
                cmVsLasr=new SensorMoverMediator("irst-Vs-CM",irsts[i],cm);
                cmVsLasr.setMinTimeStep(minTstep);
            }
    /**
    * Make and connect the mediators for the LCHR's
    */
            for(int i=0;i<lchrs.length;i++) {
                SensorMoverMediator cmVsLchr;
                cmVsLchr=new LauncherMoverMediator("lchr-Vs-CM",lchrs[i],cm);
                cmVsLchr.setMinTimeStep(minTstep);
            }
            for(int i=0;i<lasrs.length;i++) {
                lasrs[i].addModEventListener(tc);
            }
            for(int i=0;i<lchrs.length;i++) {
                lchrs[i].addModEventListener(fd);
            }
            for(int i=0;i<irsts.length;i++) {
                irsts[i].addModEventListener(tc);
            }
            for(int i=0;i<numMissiles;i++) {
                ffc.reset();
                tc.reset();
                Schedule.reset();
                //use the four lasrs and the airbase as targets.
                Route4D r=IADSutils.randomRoute2( new Coor3D[] {new
    Coor3D(110,125,0)});
                cm.setRoute(r);
                cm.go();
                Schedule.startSimulation();
                System.out.println(
    IRSTrange+"\t"+LASRrange+"\t"+LCHRrange+"\t"+ ffc);
            }

        public static void main(String[] args) {
            System.out.println("IRST"+"\t"+"LASR"+"\t"+"LCHR"+"\t"+"FFC");
            for(int ir=0;ir <= 20;ir+=2) {
                for(int la=0; la<=20; la+=2) {
                    simulate(100,ir,la,15.0,0.1);
                }}}}
```

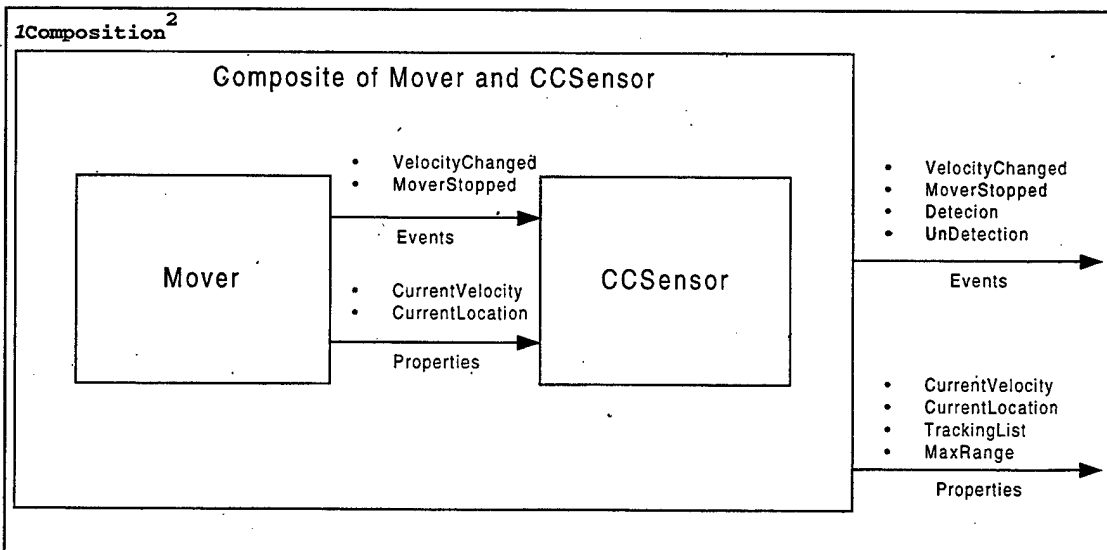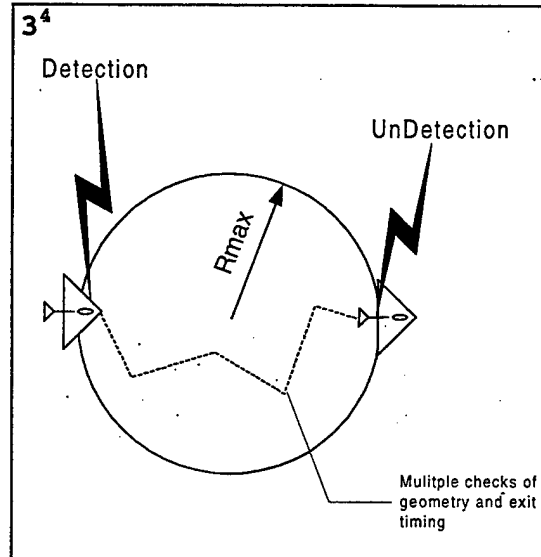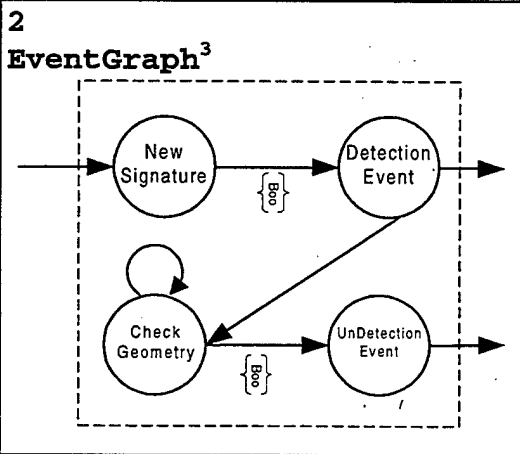# APPENDIX E.   COMPONENT LIBRARY FACT-SHEETS

This appendix provides fact-sheets for two of the components in the Modkit component library used in this thesis. The two components are the BasicSensor, the three-dimensional cookie-cutter sensor, and the RouteMover, the component forming the basis for the simulated cruise-missile and surface-to-air missile.

Modkit Component Fact-Sheet

Name: CCSensor **Category:**[1] Component

Description:
The CC Sensor is handed a new signature from a mediator. If the signature is within max range it is considered detected and the CCSensor fires a detection event. Then it schedules an update of the target, using the targets current position and velocity vector to deduce when the target may become lost (exit the detection volume). If the targets velocity vector changes before the check is performed, the scheduled check is cancelled and a new one scheduled according to the new velocity vector. The target can be any composite, but must contain a component that can provide the current location and current velocity vector of the target. The sensor itself does not model its own position or movement in space. If the CCSensor is not connected to a mover, then it will simply be a static sensor, with location as set in the default location.

2

EventGraph[3]

3[4]

Detection

UnDetection

Rmax

Mulitple checks of geometry and exit timing

1Composition[2]

Composite of Mover and CCSensor

Mover

- VelocityChanged
- MoverStopped

Events

- CurrentVelocity
- CurrentLocation

Properties

CCSensor

- VelocityChanged
- MoverStopped
- Detecion
- UnDetection

Events

- CurrentVelocity
- CurrentLocation
- TrackingList
- MaxRange

Properties

Modkit Component Fact-Sheet

Interfaces

Incoming

Events Handeled

EventID	EventClass[5]	Action performed
NewSignature	NewSignatureEvent	Tests if detection possible, if so fires detection event and schedules update of geometry for undetection event
MoverStopped	MoverStoppedEvent	If target (mover) is tracked then interrupts all future actions on this target
VelocityChanged	VelocityChangedEvent	This triggers recalculation of geometry since time to undetect have changed

Properties Required

Name	Class	Usage	Default
CurrentLocation	Coor4D	To calculate distance to target	Settable
CurrentVelocity	Coor3d	To calculate undetection in future	(0,0,0)
MaxRange	Double	Targets withing this range are detected	0

Outgoing

Events Generated

EventID	EventClass	On Condition
Detection		
UnDetection		

Data/Properties Provided

Name	Class	Default
MaxRange	Double	0
TrackingList.	Vector	Empty

Syntactic

Standard[6]

75

Modkit Component Fact-Sheet

Compatible Mediators

Name	Mediates in composite[7]
MoverSensorMediator	Yes

Hints and Tips for Usage

Special considerations using this component

Modkit Component Fact-Sheet

Construction

Connection

Java Doc

Java Source Code

```java
/**
* @author Arent Arntzen
* @version 0.1
* Started 5 Jun 98
*/


package modsim;
import modkit.*;
import modutil.spatial.*;
import simkit.*;

public class BasicSensor extends BasicModSimComponent {
    protected Coor4D       targetLocation;
    protected Coor4D       sensorLocation;
    protected double       maxRange;
    protected double       rangeBuffer;
    protected double       distToTarget;
    protected ModComponent trackedTarget;
    protected int          maxTrackedTargets;
    protected double       targetMaxSpeed;
    protected double       sensorMaxSpeed;
    protected Coor3D       sensorCurrentVelocity;
    protected Coor3D       targetCurrentVelocity;
    protected double       relativeSpeed;
    protected double       minTimeStep;
    protected double       detectionTime;
    protected double       unDetectionTime;
    protected boolean      inRange=false;

    public BasicSensor(boolean introspect) {
        super(introspect);
        if (introspect) {
            propertyDispatcher=new PropertyDispatcher(this);
            eventDispatcher=new EventDispatcher(this);
        }
        maxRange=0.0;
        rangeBuffer=0.01;
    }

    public BasicSensor(String  name,
                       boolean introspect) {
        this(introspect);
        setName(name);
    }
```

```java
public BasicSensor(String  name) {
    this(true);
    setName(name);
}

public void addModPropertySource(ModPropertySource modPropertySource) {
    super.addModPropertySource(modPropertySource);
    init();
}

public BasicSensor() {
    this(true);
}

public void setMaxRange(double mr) {
    maxRange=mr;
}

public double getMaxRange() {
    return maxRange;
}

public void setMaxRangeBuffer( double fract) {
    rangeBuffer=fract;
}

public int getMaxTrackedTargets() {
    return maxTrackedTargets;
}

public void setMaxTrackedTargets(int max) {
    maxTrackedTargets=max;
}

private void upDateSensorLocation() {
    sensorLocation=(Coor4D)getProperty("CurrentLocation",new
Coor4D(0,0,0,0));
}

private void upDateTargetLocation() {
    targetLocation=(Coor4D)
trackedTarget.getProperty("CurrentLocation",new Coor4D(0,0,0,0));
}

private void upDateTargetMaxSpeed() {
    targetMaxSpeed=((Double)trackedTarget.getProperty("MaxSpeed",new
Double(0))).doubleValue();
}

private void upDateRelativeSpeed() {

targetCurrentVelocity=((Coor3D)trackedTarget.getProperty("CurrentVelocity",
    new Coor3D(0,0,0)));
    sensorCurrentVelocity=((Coor3D)getProperty("CurrentVelocity",new
Coor3D(0,0,0)));

relativeSpeed=sensorCurrentVelocity.sub(targetCurrentVelocity).norm();
}
```

```
    public void doCheckGeometry() {
        upDateTargetLocation();
        upDateSensorLocation();
        distToTarget=sensorLocation.distTo(targetLocation);
        if (distToTarget > getMaxRange()) {
          inRange=false;
          generateUnDetectionEvent();
        }
        double timeToNextCheck=(getMaxRange()-
        distToTarget)/(targetMaxSpeed+sensorMaxSpeed);

minTimeStep=(getMaxRange()*rangeBuffer)/(targetMaxSpeed+sensorMaxSpeed);
        if (timeToNextCheck < minTimeStep) {
            timeToNextCheck=minTimeStep;
        }
        if (inRange) {
            waitDelay("doCheckGeometry",timeToNextCheck);
        }
    }

    public void handleNewSignatureEvent(ModEvent e) {
        NewSignatureEvent nse=(NewSignatureEvent) e;
        ModComponent tgt=(ModComponent) nse.getSignature();
        targetLocation=(Coor4D) nse.getLocation();
        upDateSensorLocation();
        distToTarget=sensorLocation.distTo(targetLocation);
        if (distToTarget <= getMaxRange()) {
            inRange=true;
            trackedTarget=tgt;
            upDateTargetMaxSpeed();
            generateDetectionEvent();
        }
    }

    public void generateDetectionEvent() {
        detectionTime=Schedule.simTime();
        ModEvent e=new
DetectionEvent(this,trackedTarget,targetLocation,sensorLocation);
        notifyListeners(e);
        waitDelay("doCheckGeometry",0.0);
    }

    public void generateUnDetectionEvent() {
        unDetectionTime=Schedule.simTime();
        inRange=false;
        interruptAll();
        ModEvent e=new
UnDetectionEvent(this,trackedTarget,targetLocation,sensorLocation,
        unDetectionTime-detectionTime);
        notifyListeners(e);
    }

    public void handleMoverStoppedEvent(ModEvent e) {
        interruptAll();
    }
}
```
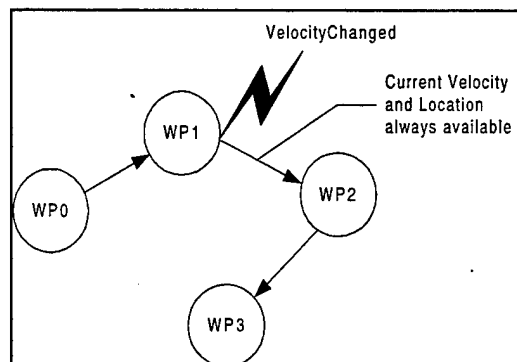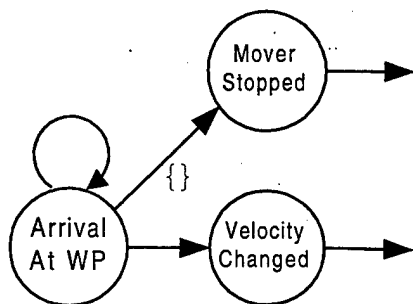
Modkit Component Fact-Sheet

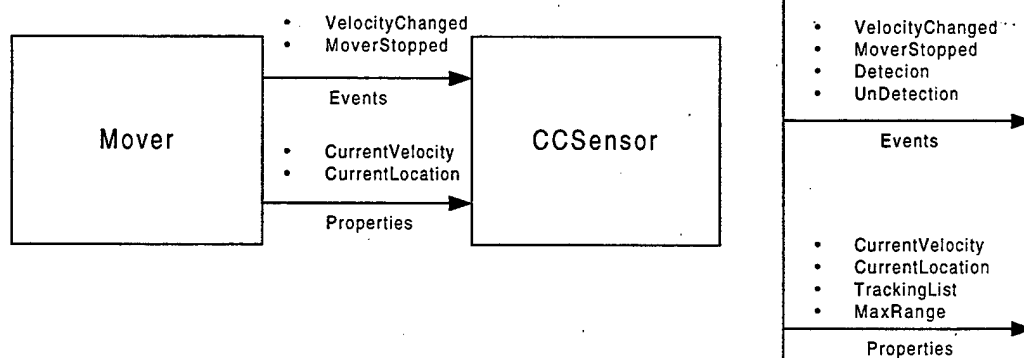Name: **RouteMover** Category:[8] **Component**

Description:[9]
The mover moves from waypoint to waypoint with constant velocity between waypoints.
Waypoints and velocity are 3 dimensional.Locations are 4 dimensional with time as the
forth coordinate. Upon arrival at a waypoint it fires a VelocityChanged event. At its
final destination it fires a MoverStopped event. CurrentLocation and Velocity is always
available by using the getProperty method. Since motion is assumed to be linear
translation between waypoints, location at any time is calculated by using linear
interpolation between points.

4EventGraph[10][11]

Mover Stopped

{}

Arrival At WP

Velocity Changed

VelocityChanged

WP1

Current Velocity and Location always available

WP0

WP2

WP3

5Composition[12]

Composite of Mover and CCSensor

Mover

- VelocityChanged
- MoverStopped

Events

- CurrentVelocity
- CurrentLocation

Properties

CCSensor

- VelocityChanged
- MoverStopped
- Detecion
- UnDetection

Events

- CurrentVelocity
- CurrentLocation
- TrackingList
- MaxRange

Properties

80

Modkit Component Fact-Sheet

Interfaces

Incoming

Events Handeled

EventID	EventClass[13]	Action performed
KillRemove	KillRemoveEvent	Deregisters with all listeners Marks itself for disposal

Properties Required

Name	Class	Usage
Route	Route4D	Initialize to this route

Outgoing

Events Generated

EventID	EventClass	On Condition
VeloctiyChanged	VelocityChangedEvent	When arriving at next waypoint
MoverStopped	MoverStoppedEvent	Upon arrival at final WP

Data/Properties Provided

Name	Class	Default
CurrentLocation	Coor4D	(0,0,0,0)
CurrentVelocity	Coor3D	(0,0,0)

Syntactic

Standard[14]

Modkit Component Fact-Sheet

Compatible Mediator

Name	Mediates in composite[15]
MoverSensorMediator	Yes

Hints and Tips for Usage

Initialize by passing the mover a route. See Java code for details. The route can be made in different ways, giving you some flexibility.

Modkit Component Fact-Sheet

Code

Construction

Connection

Java Doc

Java Source Code

```java
/**
* @author Arent Arntzen
* @version 0.1
* Started 31 May 98
*/

package modsim;

import modkit.*;
import modutil.spatial.*;
import simkit.*;

public class RouteMover extends BasicModSimComponent {
    protected Route4D route;
    protected Coor4D lastPosition
    protected Coor3D lastVelocity
    protected double maxSpeed;

    public RouteMover() {
        super(false);
        propertyDispatcher=new PropertyDispatcher(this);
        eventDispatcher=new EventDispatcher(this);
        lastVelocity=new Coor3D(0,0,0);
    }

    public RouteMover(String name) {
        this();
        setName(name);
    }

    public void setMaxSpeed(double speed) {
        maxSpeed=speed;
    }

    public double getMaxSpeed() {
        return maxSpeed;
    }

    public void setRoute( Route4D r) {
        route=r;
        lastPosition=route.peekNextWP();
    }

    public Route4D getRoute() {
```

```
        return route;
    }


public Coor4D getCurrentLocation() {
        double atTime=Schedule.simTime();
        double deltaTime=atTime-lastPosition.getT();
        Coor3D deltaMove=(Coor3D)lastVelocity.scalarMul(deltaTime);
        Coor3D lPos=new Coor3D(lastPosition);//make 3D version of lastpos
        Coor3D newPos=(Coor3D) lPos.add(deltaMove);//find new 3D pos
        return new Coor4D(newPos,lastPosition.getT()+deltaTime);//return 4D
version
    }


public Coor3D getCurrentVelocity() {
        return lastVelocity;
    }


public double getCurrentSpeed() {
        return lastVelocity.norm();
    }


private void goToNextWP() {
        if (route.hasMoreWP()) {
            waitDelay("doArrivalAtLocation",route.getTimeToNextWP());
        }
        else {
            ModEvent e=new MoverStoppedEvent(this);
            notifyListeners(e);
        }
    }

public void doArrivalAtLocation() {
        Coor4D newPos=route.getNextWP();//since this method is called now we
have arrived
        lastPosition=newPos; //so now this becomes the last known position
        lastVelocity=route.getVelocityToNextWP();//and this the last known
velocity vector
        ModEvent e= new ArrivalAtLocationEvent(this,newPos,lastVelocity);
        //(this,newPos,lastVelocity);
        notifyListeners(e);
        goToNextWP();//Now schedule the next arrival
        generateVelocityChangedEvent();
    }

    public void generateVelocityChangedEvent() {
        ModEvent e=new VelocityChangedEvent(this, lastVelocity);
        notifyListeners(e);
    }

public void go() {
        goToNextWP();
    }
}
```

Modkit Component Fact-Sheet

Notes

---

[1] Can be Component or Mediator
[2] Shows example of composition with connections established
[3] Eventgraph for Discrete Event Simulation Component or Mediators
[4] Shows example of composition with connections established
[5] This must correspond to the Java classname of the eventobject
[6] The standard wiring interfaces defining a ModComponent
[7] Yes if mediator works when this component is embedded in composite
[8] Can be Component or Mediator
[9] The picture box optionally graphically describes the component
[10] Unconnected arrows are incoming or outgoing shared events
[11] Eventgraph for Discrete Event Simulation Component. or Mediators
[12] Shows example of composition with connections established
[13] This must correspond to the Java classname of the eventobject
[14] Standard syntactic interfaces are ModEvent, ModEventListener, ModEventSource, PropertyProvider, PropertyUser
[15] Yes if mediator works when this component is embedded in composite

INITIAL DISTRIBUTION LIST